Position Paper: The Collapse Macro in Best-First Search Algorithms and an Iterative Variant of RBFS

Ariel Felner
Information Systems Engineering
Ben-Gurion University
Be'er-Sheva, Israel 85104
felner@bgu.ac.il

Abstract

This paper makes two pedagogical contributions. First, we describe two macro operators for best-first search algorithms: the *collapse* macro where a subtree is deleted from memory and its best frontier value is stored in its root, and, the *restore* macro (the inverse of *collapse*) where the subtree is restored to its previous structure. We show that many known search algorithms can be easily described by using these macros. The second contribution is an algorithm called *Iterative Linear Best-first Search* (ILBFS). ILBFS is equivalent to RBFS. While RBFS uses a recursive structure, ILBFS uses the regular structure of BFS with occasionally using the collapse and restore macros. ILBFS and RBFS are identical in the nodes that they visit and have identical properties. But, I believe that ILBFS is pedagogically simpler to describe and understand; it could at least serve as a pedagogical tool for RBFS.

Introduction

Best-first search (BFS) is a general scheme. It seeds an *open list* (denoted as OPEN) with the start state. At each *expansion cycle* the most promising node (*best*) from OPEN is removed and its children are generated and added to OPEN while *best* is moved to a *closed list* (denoted as CLOSED). Special cases of BFS differ in their cost function f. For example, A* is a BFS with $f(n) = g(n) + h(n)$, where $g(n)$ is the sum of the edge costs from the start to n and $h(n)$ is a heuristic estimation of the cost from n to a goal.

The main structure of BFS, i.e, always expanding a node on OPEN with minimal f-value ensures that under some conditions (known as cost algebra) (Stern et al. 2014; Edelkamp, Jabbar, and Lluch-Lafuente 2005) it has a number of optimal properties. For example, that A* with an admissible heuristics returns the optimal path and is optimally effective. Therefore, many search algorithms use this BFS structure but they add enhancements or modifications on top of this structure. The list of relevant algorithms is so large that we omit a list of references.

This paper makes two pedagogical contributions. First, it describes two macro operators for BFS algorithms: *collpase* and *restore*. A collapse action occurs at a closed node r. The subtree below r is removed from both OPEN and CLOSED and r is moved to OPEN along with the minimal f-value among the nodes of the frontier of the deleted subtree. Later, a *restore* macro can be applied to restore the subtree to its previous structure. We show that many known search algorithms (e.g., IDA* (Korf 1985), SMA* (Russell 1992), LookaheadA* (Stern et al. 2010) and EPEA* (Goldenberg et al. 2014)) can be easily described by the BFS scheme plus different ways of using the collapse and restore macros.

Recursive Best-First Search (RBFS) (Korf 1993) is a fascinating algorithm which is strong competitor to IDA*. However, unlike IDA*, RBFS is very complicated to grasp, especially for new comers. One needs to spend quite some time in order to fully understand RBFS and appreciate its beauty. This is probably one of the reasons that in practice many implementers and researchers refrain from using RBFS and it is not as commonly used as IDA*. The second contribution of this paper tries to remedy this by introducing an algorithm called *Iterative Linear Best-first Search* (ILBFS). ILBFS is a variant of RBFS which uses the regular structure of BFS but also uses the collapse and restore macros. ILBFS and RBFS are identical in the nodes that they visit. But, I believe that ILBFS is much simpler to describe and understand and that ILBFS could serve as a pedagogical tool for easily understanding RBFS. Hopefully, ILBFS will cause more researchers and practitioners to understand and consider to use RBFS.

The collapse and restore macros

In this section we focus on two techniques which are used by different search algorithms in different contexts but were never spelled out on their own. We call these general techniques: *collapse* and *restore* of frontier nodes.

A known invariant in best-first search algorithms is that OPEN is a perimeter of nodes around the start state. We call this the *perimeter invariant*. This invariant implies that at all times during the course of the algorithm (i.e., before the goal node was chosen for expansion), there exists a node in OPEN along any path to the goal. This invariant is easily proven by induction. It is initially true when OPEN only includes that start state. The induction step is on the *expand* action. Assume a node x was selected for expansion. While it is removed from OPEN, all its successors are added to OPEN.[1] Let P be a path to the goal and let $n \in P$ be a

[1] We assume that in case of inconsistent heuristics closed nodes

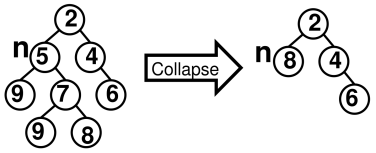

Figure 1: collapsing the frontier

node in OPEN before the expansion of x. If $x \neq n$ then n is still in OPEN after the expansion cycle. If $x = n$ then n is now removed from OPEN. Let n' be the successor of n in P. $n' \in P$ was just added to OPEN.

This invariant can be generalized to the *subtree invariant*. Let r be a node that was generated by the best-first search algorithm. At any point of time thereafter, a complete cut of the subtree rooted at r must be present in OPEN. The perimeter invariant above is a special case of the subtree invariant where r is the initial node of the search.

This invariant can be used in a reverse order too. Any set of frontier nodes in OPEN can be removed and replaced by their common ancestor (or ancestors) and the invariant still holds. This entails the *collapse* macro defined next.

The collapse macro

For simplicity we assume that all nodes in OPEN and CLOSED have parent pointers and that all nodes in CLOSED also have pointers to their children.

Let r be a node in CLOSED and let $T(r)$ be the subtree rooted at r that includes all of r's descendants that were already generated by the algorithm. Note that the internal nodes, including r itself, of this subtree, denoted by $T_i(r)$, are in CLOSED while the frontier nodes of this subtree, denoted by $T_f(r)$, are in OPEN.[2] The $collpase(r)$ macro is defined as follows:

(1) $T_i(r)$ is removed from CLOSED.

(2) $T_f(r)$ is removed from OPEN.

(3) r is added to OPEN with the best f-value in $T_f(r)$.

(4) A binary variable $r.C$ is set to $True$ to denote that r is on OPEN but it is a root of a subtree that was collapsed. This means that the f-value of r was propagated from one of its descendants. This variable is optional. In some cases (as described below) this variable is not used.

The collapse macro is illustrated in Figure 1 where the entire left subtree rooted at n is collapsed into n. $f(n)$ is now changed to 8, the minimal f-value among all the collapsed leaves. In addition, $n.C$ is set to $True$.

We borrow the terminology first used in RBFS (Korf 1993). We use $f(n)$ (small f) and the term *static value* to denote the regular f-value of a node n. The minimal f-value of the leaves propagated up to a common ancestor n by a collapse action is called the *stored value* of n and is denoted by $F(n)$ (capital F). In Figure 1, the static value of n is $f(n) = 5$ while its stored value is $F(n) = 8$. When the search reaches a node n in OPEN with $F(n) \neq f(n)$, we know that n has been expanded before and that its stored value is the minimal frontier node below n. But, in general, in some cases it might be that $F(n) = f(n)$. For this reason we use the binary variable $n.C$ to denote whether the stored value $F(n)$ is a result of a collapse action or not.

We note that the collapse macro is a general idea. It may be implemented or partially implemented in many ways.

The restore macro

The *restore* macro is the inverse of collapse. It can be applied on a node r from OPEN where $r.C = True$. In this case, we restore the same subtree that was collapsed back into OPEN and CLOSED. Collapse is a form of lossy compression. Therefore, the main question that arises is whether we can identify the exact same subtree that was collapsed only by the F-value stored at r. There is no definitive answer to this and this is algorithm/domain dependent.

A simple, special case for *restore* is when the f-value of nodes is *monotonically non decreasing* along any path. This happens for example, when the heuristic h is *admissible and consistent*. In this case, we can grow a tree below r as long as we see f-values that are *smaller than* $F(r)$. This can be done, e.g., with a depth-first search (DFS) below r. In this DFS when a node n is generated with $f(n) < F(r)$ it is expanded and added to CLOSED. When a node n is generated with $f(n) \geq F(r)$ then n is added to OPEN with $f(n)$, $n.C$ is set to $False$ and the DFS then backtracks from n. This ensures that we restore the subtree below r such that the minimal f-value in $T_f(r)$ is $F(r)$.

The restore macro is more problematic when there is no guarantee that the f-value is *monotonically non decreasing* along paths, for example, when using $f(n) = g(n) + W \times h(N)$ for $W > 1$ (as in WA*). Here, if we stop the DFS below r when we reach a node m with $f(m) > F(r)$ we have no guarantee that we are in the frontier of the collapsed tree as there might be descendants of m with smaller f-values. However, there are cases where the restore macro is fully applicable even without the monotonicity property. This happens for example for both ILBFS and RBFS described below. These algorithm activate the restore macro in such a way that the old collapsed tree is fully restored (see below).

Known algorithms using the macros

The *collapse* and *restore* macros are used by many algorithms, many of which can be easily described with these macros. We cover such algorithms in this section.

IDA*

IDA* can be seen as performing restore and collapse actions on the root node as follows. First, $h_0 = h(root)$ is calculated. Then, $restore(root, h_0)$ is called where we activate the DFS method described above.[3] Then, until the goal is found, we loop as follows. We call $Collpase(root)$ and then we call $restore(root, F(root))$.[4]

are *re-opened* and then *re-expanded*.

[2]For simplicity, we assume here that this is the first time that the *collapse* macro is activated.

[3]Logically, this not an actual restore because that tree was never available. But, this is indeed a DFS with the relevant bound.

[4]It is important to note that IDA* follows a best-first expansion order (i.e., generates new nodes across iterations in a best-first order according to $f = g+h$.) only if the cost function is monotonic.

SMA*

Simplified Memory-bounded A* (SMA)* (Russell 1992) is a memory efficient variant of A*. It performs basic A* cycles until memory is full. In this case, it can only expand a node and add a node to OPEN if another node is deleted. To free space, the node in OPEN with the worst f-value is chosen and deleted. But, at this point, a *collapse* action is called on its parent so that the best value of the parent's children is now stored.[5] When a root r of a collapsed subtree is chosen for expansion then SMA* expands and generates the nodes below it one by one. However, it uses the *pathmax method* to pass $F(n)$ from parents to children along that subtree. This can be seen as a restore macro.

A* with lookahead

A with lookahead* (AL*) (Stern et al. 2010; Bu et al. 2014) uses the same basic expansion cycle of A*. The least cost node n from OPEN is expanded and moved to CLOSED, and its children are generated and added to OPEN. After a node n is expanded, a limited lookahead is first performed from n by using a DFS. This DFS is bounded by a parameter K which allows the DFS to continue as long as it reaches nodes m for which $f(m) \leq f(n) + K$. This DFS can be seen as a restore action. Then, f-values are backed up to n and this can be seen as a collapse action on node n. In AL*, again we perform a restore action before a collapse action.

Partial expansion A*

*Partial expansion A** (PEA*) (Yoshizumi, Miura, and Ishida 2000) and its enhanced version EPEA* (Felner et al. 2012; Goldenberg et al. 2014) also uses the collapse macro. When expanding node n, PEA*/EPEA* first generates a list $C(n)$ of all the children of n. However, only the subset of $C(n)$ with f-values that are equal to $F(n)$ are added to OPEN. The rest children are collapsed into node n and n is added back to OPEN with the smallest f-value among the children. No restore macro is used here.

RBFS

Recursive best-first search (RBFS) (Korf 1993) is a linear-space algorithm which simulates best-first search expansions. In this sense, it is a strong competitor to IDA* (Korf 1985). The main advantage of RBFS over IDA* is that in RBFS new nodes are expanded in a *best-first order* even if the f-function is not *monotonically increasing*. By contrast, IDA* expands new nodes in a best-first only for *monotonically increasing* f-functions. Due to their linear-space complexity both algorithms suffer from the same problem of not being able to detect duplicates. Experimental results and analytical studies show pros and cons in the performance of these algorithms and neither of these can be declared as a winner over the other (Korf 1993).

In practice, however, IDA* is massively used by researchers and is considered the benchmark algorithm for performing linear-space BFS. According to Google Scholar (April 28, 2015), IDA* is cited 1572 times while RBFS is cited 328 times . An enormous number of research papers use, study or enhance IDA* while RBFS is rarely used or studied. Neller (2002) used RBFS as its leading algorithm while Hatem, Kiesel and Ruml. (2015) recently published a paper that provides an enhancement to RBFS. But, these are rare exceptions compared to the number of papers published on IDA*. What is the reason for this gap?

W. Ruml, one of the authors of (Hatem, Kiesel, and Ruml 2015) said in his talk at AAAI-2015: *"for many years, I was afraid of RBFS"*. His was afraid by the fact that RBFS may bounce back and forth from different areas of the search tree. Indeed, Hatem et al. provided a technical enhancement to RBFS that remedies this behavior.

The author of this paper believes that the reason that researchers refrain from using RBFS and use IDA* instead, is not necessarily the disadvantages of RBFS (as discussed and addressed by Hatem et al. (2015) but its pedagogical complexities. The pedagogical difficulties of understanding IDA* and RBFS are in opposite extremes. IDA* is very easy to understand and to implement even for beginners. A few explanations together with a simple running example is enough for seeding this algorithm in a classroom of students. By contrast, RBFS is very complicated. Even the most talented teacher will need several hours to teach it and no student will be able to fully grasp its elegant behavior and appreciate its beauty without reading the explanations on RBFS a few more times and performing a decent number of practice runs on arbitrary example trees. In fact, R. E. Korf who invented both IDA* and RBFS realized that RBFS is pedagogically complex and said that: *"most people do not understand it"*.[6]

Iterative linear best-first search

To remedy this, based on the macros defined above, I would like to introduce an algorithm called *Iterative linear best-first search* (ILBFS). ILBFS is fully identical to RBFS in the nodes it visits but its logical structure is different. Instead of describing RBFS and then moving to ILBFS, we will do the opposite as ILBFS is easier to understand in my opinion. Then, one may move to its recursive variant namely RBFS. ILBFS simulates RBFS using the classic best-first search framework, i.e., it maintains an OPEN and CLOSED list and essentially performs a series of expansion cycles. Nevertheless, in order to ensure the *linear space* requirement, it heavily uses the *collapse* and *restore* macros. I believe that pedagogically, ILBFS should be taught/explained first.

Best node chosen for expansion

Before moving to ILBFS, we first present an important observation that we will need below.

In a regular implementation of BFS the currently known search tree is stored in memory at all times, where all the internal nodes are in CLOSED and all the frontier nodes are in OPEN. Let $oldbest$ be the node just expanded by BFS, i.e., $oldbest$ was deleted from OPEN and its b children were just

This is due to the exact the same reason just described for a restore macro that is implemented by a bounded DFS.

[5] In fact, SMA* performs partial collapse actions (called *backup* in that paper) every time a node is generated.

[6] Personal communication with the author (Summer 1998).

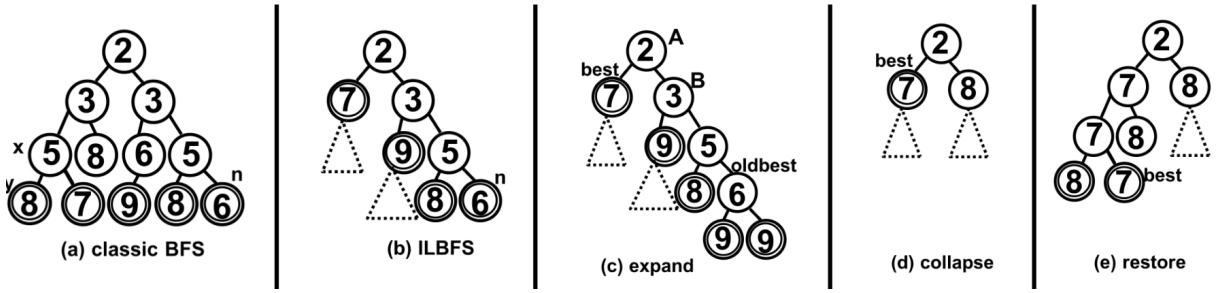

Figure 2: Best-first search tree and how ILBFS develops it

added to OPEN (where b is the branching factor). Let $best$ be the new best node in OPEN. There are now two possible cases with regards to the identity $best$:

- **Case 1:** $best$ is a child of $oldbest$. In this case, BFS logically operates like a DFS. The search deepens one level below $oldbest$ and adds the children of $best$ to the tree.
- **Case 2:** $best$ is not a child of $oldbest$ but is somewhere else in the tree. BFS now moves to another location of the tree and performs the expansion operation at that location. But, $oldbest$ is kept in CLOSED and its children are kept in OPEN. This behavior contributes to the exponential growth of OPEN (with respect to the depth of the tree) as at each point of time one node is deleted from OPEN but b nodes are added instead, without necessarily increasing the depth of the tree.

Tree of ILBFS

We now move to explaining ILBFS. Regular BFS stores the entire search tree in memory. By contrast, ILBFS only stores one branch of internal nodes plus their immediate children. Let *principal branch* denote the branch of internal nodes that includes all ancestors of $best$. ILBFS stores the principal branch and their immediate children, i.e., ILBFS stores the branch that has *best* in its bottom level. In addition, the fact that only this branch can be stored in memory is called the *principal branch invariant*.

Figure 2(a) presents a typical tree where leaf nodes (double framed) are in OPEN. The rightmost node n is $best$. Figure 2(b) presents the way ILBFS stores this tree. Only the principal branch (i.e., the right branch of internal nodes) and their children are stored in memory. Other parts of the tree are collapsed into children of the principal branch. Given that d is the depth of the tree and that b is the branching factor, then the amount of memory needed will be $O(b \times d) = O(d)$ assuming b is constant. Therefore, the amount of memory needed for ILBFS is linear in the depth of the search tree.

High-level description of ILBFS

ILBFS uses the regular best-first search expansion cycle and it mimics a regular BFS on the complete tree. However, it is restricted to keep the principal branch invariant and it occasionally calls collapse/restore actions as described next.

ILBFS maintains two data structures: **(1) TREE:** includes the principal branch and its immediate children. **(2) OPEN:** includes the frontier of TREE.

ILBFS needs to keep the principal branch invariant in both cases defined above for the relation between $oldbest$ and $best$ as follows:

- **Case 1:** Here, $best$ is a child of $oldbest$. In this case ILBFS is identical to regular BFS. $best$ is chosen for expansion and its children are added to OPEN. The principal branch invariant is kept but it becomes one level deeper.
- **Case 2:** $best$ is not a child of $oldbest$.[7] ILBFS cannot yet go ahead and expand $best$ (as in regular BFS) as this will violate the principal branch invariant. ILBFS overcomes this difficulty as follows. Let A be the common ancestor of $oldbest$ and $best$. Due to the principal branch invariant A must be the parent of $best$. In addition, let B be the sibling of $best$ (B is a child of A) which is also an ancestor of $oldbest$. In order to keep the principal branch invariant the subtree below B is collapsed into B.

For example, assume that node n of Figure 2(b) was expanded as $oldbest$. Its children are added to the tree (figure 2(c)). Next, the the leftmost node with $F = 7$ is the new $best$. Since $best$ is not a child of $oldbest$ we identify A the common ancestor of $best$ and $oldbest$. Similarly, we identify B, A's child which is also an ancestor of $oldbest$. We then activate the collapse macro on B as shown in Figure 2(d).

Finally, the last issue that needs to be taken care of is that sometimes a node $best$ in OPEN is chosen for expansion but its F-value was collapsed from one of its descendants. This can be detected by checking the $best.C$ variable. In this case, we apply the *restore* macro on $best$ to reveal the *real* best node from the frontier. In fact, we only want to restore the principal branch. This is shown in Figure 2(e). We deal with the low-level details of how this is done below.

The high-level pseudo code for ILBFS is presented in Algorithm 1. The basic structure of a regular best-first search is easily recognized. Lines 7–11 make sure that the principal branch invariant is kept. The *collapse* macro is called in lines 8–9. The *restore* macro is called in lines 10–11.

Low level implementation of ILBFS

We now cover low-level details of ILBFS. In particular, we will give the exact pseudo code for the collapse and restore macros used by ILBFS. To do this we provide a more detailed pseudo code of ILBFS in Algorithm 2.

[7] $best$ refers to the node currently in OPEN with the minimal F-value, even if its value was collapsed from one of its descendants.

Algorithm 1: High-level ILBFS

```
   Input: Root R
1  Insert R into OPEN and TREE
2  oldbest=NULL
3  while OPEN not empty do
4      best=extract_min(OPEN)
5      if goal(best) then
6          exit
7      if oldbest ≠ best.parent then
8          B ← sibling of oldbest that is ancestor of best
9          collapse(B)
10     if best.C=True then
11         best ← restore(best)
12     foreach child C of best do
13         Insert C to OPEN and TREE
14     oldbest ← best
```

Algorithm 2: Low-level ILBFS

```
   Input: Root R
1  Insert R into OPEN and TREE
2  oldbest=NULL
3  while OPEN not empty do
4      best=extract_min(OPEN)
5      if goal(best) then
6          exit
7      while (oldbest ≠ best.parent) do
8          oldbest.val ← min(values of oldbest children)
9          Insert oldbest to OPEN
10         Delete all children of oldbest from OPEN and TREE
11         oldbest ← oldbest.parent
12     foreach child C of best do
13         F(C) ← f(C)
14         if F(best) > f(best) and F(best) > F(C) then
15             F(C) ← F(best)
16         Insert C to OPEN and TREE
17     oldbest ← best.
```

Collapse in ILBFS

The exact implementation for the collapse macro in ILBFS is presented in Lines 7-11 of Algorithm 2. We have both $oldbest$ and $best$ at hand. The branch of $oldbest$ and its children is collapsed iteratively, bottom up, until we reach B, the sibling of the new best node. When this node is reached, the *while()* sentence is no longer true and the loop ends.

For example, in Figure 2(c) after $oldbest$ was expanded and the new best node $best$ is revealed we realize that $oldbest$ is not the parent of $best$. Therefore, the children of $oldbest$ are collapsed into $oldbest$ and $oldbest$ is placed back in OPEN (Line 9). Next we do the same for its parent and continue this up the branch until we reach node B which is the sibling of the new $best$ node.

Restore in ILBFS

Performing the restore action is challenging and perhaps the only complex pedagogical step in ILBFS. All we have at hand is a node $best$ together with its stored value $F(best)$. How do we re-generate the principal branch to its structure before the collapse action?

Here we rely on an observation that was first seen by Korf in the original RBFS paper (Korf 1993). We repeat it here in the context of ILBFS.

Observation 1: In ILBFS after a *collapse* action is applied on a node B then $F(B) > f(B)$.

Proof: Let $oldbest$ and $best$ be the best nodes chosen for expansion in successive cycles and let B be the ancestor of $oldbest$ (the sibling of $best$) for which we activated the collapse action. Since $best$ is in OPEN and since only the principal branch is stored, then there must have existed a point of time where both $best$ and B were in OPEN for the first time (after expanding their ancestor). Then, B was chosen for expansion which means that $f(B) \leq F(best)$. $best$ was always in OPEN until a point where all nodes in the frontier below B have f-values larger than $F(best)$ (assuming we break ties in a depth-first order). At this point $best$ is chosen for expansion and everything below B is collapsed to $F(B) > F(best) \geq f(B)$. □

Above, we defined a binary variable $n.C$ that says whether node n is a root of a collapsed subtree or not. This was a general definition. Here, for ILBFS (and for RBFS as well) this binary variable is not needed because we can make this distinction solely based on Observation 1 as follows:

Observation 2: Assume $best$ is chosen for expansion. If $F(best) > f(best)$ then based on Observation 1 (applied on $best$), we know that $best$ is a root of a collapsed subtree whose best frontier node had f-value of $F(best)$. Otherwise, (if $F(best) = f(best)$) it is on the frontier.[8]

When $F(best) > f(best)$ we must therefore apply the restore macro. Since $F(best)$ is the minimal frontier value collapsed from the children of $best$, then we know that every child below $best$ was at least as large as $F(best)$. Therefore, during the restore process on $best$ we can propagate $F(best)$ to its child c if $F(best) > f(c)$. We call this the *restore propagation rule* which is similar to the *pathmax* method (Felner et al. 2011). In Figure 2(e) we propagate the value of 7 from parents to children below them according to this rule.

Now we are interested to reach a node below $best$ that was on the frontier with f-value of $F(best)$. For this we just perform a depth-first search below $best$ where we set the bound of the DFS to $F(best)$. During the DFS, if we reach a node n with $f(n) > F(best)$ we backtrack. If $f(n) < F(best)$ we generate all children of n. For each child c we set $F(c) = max(F(best), f(c))$ to keep the restore propagation rule. This DFS continues until we reach a node n with $f(n) = F(best)$. This means that we have reached the old frontier. At this point we have a new principal branch whose bottom level include the actual best node from the restored frontier. Then, ILBFS continues as usual.

[8]The case where $F(best) < f(best)$ will never occur as we seed $F(best) = f(best)$ and it never decreases (see pseudo code below). It is also important to note that Observations 1 and 2 are valid even in the case where the f-cost function is non-monotonic.

	BFS	SMA*	ILBFS	AL*	EPEA*
Memory	b^d	M	$b \times d$	b^{d-k}	b^{d-1}
Collapse	No	lazily	eagerly	always	always

Table 1: Continuum of algorithms

In fact, the *restore* phase can be performed by just continuing ILBFS as usual as long as we apply the restore propagation rule. This process is shown in lines 12–15 of Algorithm 2 for a given step. The main *while()* loop continues according to regular BFS and the new best node will be the best child of $best$ after applying the propagation rule (lines 14–15). We will deepen in the tree along one branch until we get to a node n where $f(n) = F(best)$. This is a node on the frontier and new nodes will be revealed below it.

Continuum of algorithms

We now put three of the BFS algorithms discussed so far into a continuum according to their memory needs and the frequency and performing collapse and restore on ancestors of nodes in the frontier. These algorithms are the leftmost three columns of Table 1. Regular BFS does not apply the macros and needs b^d memory. SMA*, consumes at most M memory (size of memory available) and performs the macros lazily when memory is exhausted. ILBFS performs the macros eagerly, every time the best node is not a child of the old best node, as described above. Thus, ILBFS needs the least amount of memory among the three algorithms – $b \times d$. In fact, we might activate ILBFS but apply the collapse macro less strictly. In this case we will have more parts of the search tree in memory and get closer to SMA*.

AL* and EPEA* are also shown in this table. Both AL* and EPEA* activate the collapse and restore macros at every node expansion for a fixed depth (k for AL* and 1 for EPEA*) regardless of the memory available. This is done in a manner that is different than the other three algorithms which basically try to collapse non-promising nodes. We thus do not see AL* and EPEA* as part of this continuum.

RBFS and its relation to ILBFS

Now after we presented ILBFS, we can pedagogically move to its recursive version, RBFS. Again, we note that RBFS was invented first but we believe that after ILBFS is well understood, RBFS will be much easier to understand compared to understanding RBFS from scratch.

It is well known that DFS can be implemented iteratively using an OPEN list or recursively via the recursion stack. A similar relation occurs between ILBFS and RBFS. RBFS performs exactly the same steps as ILBFS. ILBFS always stores the principal branch and its children in OPEN and CLOSED. RBFS uses recursive calls. The same nodes are kept in memory albeit in the recursion stack.

The key point to understand RBFS and ILBFS is the fact that in order to preserve a best-first order, all that is needed is to keep track of two nodes, the best node in OPEN ($best$) and the second best node in OPEN ($best2$). We expand $best$ and generate its children. The next best node is either $best2$ or one of the children of $best$ who becomes the new $best2$. The main difference between ILBFS and RBFS is how these values are stored. In ILBFS, $best2$ is naturally stored in OPEN.[9] By contrast, RBFS is recursively applied on a node with an upper bound which equals to the second best node that was generated so far in the tree. This is a *virtual* bound that is updated on the fly according to some formulas given in RBFS. This makes RBFS much harder to understand.

RBFS(n, B)
1. if n is a goal
2. $\quad solution \leftarrow n$; exit()
3. $C \leftarrow expand(n)$
4. if C is empty, return ∞
5. for each child n_i in C
6. $\quad$ if $f(n) < F(n)$ then $F(n_i) \leftarrow max(F(n), f(n_i))$
7. $\quad$ else $F(n_i) \leftarrow f(n_i)$
8. $(n_1, n_2) \leftarrow best_{\mathrm{F}}(C)$
9. while ($F(n_1) \leq B$ and $F(n_1) < \infty$)
10. $\quad F(n_1) \leftarrow$ RBFS($n_1, min(B, F(n_2))$)
11. $\quad (n_1, n_2) \leftarrow best_{\mathrm{F}}(C)$
12. return $F(n_1)$

Figure 3: Pseudo-code for RBFS.

We now describe RBFS and recognize the collapse and restore macros hidden behind the recursive structure. Instead of providing our own description we borrow the description of RBFS and its pseudo code from the most recent paper on RBFS (Hatem, Kiesel, and Ruml 2015).[10] Our remarks are given in brackets in **bold** fonts.

Pseudo-code for RBFS is shown in Figure 3. Its arguments are a node n to be explored and a bound B that represents the best f value of an unexplored node found elsewhere in the search space so far. **[Our remark: node n is $best$ while B is the F-value of $best2$. In ILBFS, we have $best2$ at hand.]** Each generated child node is given an f value, the usual $g(n_i) + h(n_i)$, and an F value, representing the best known f value of any node below n_i that has not yet been expanded. The F value of a child node is set to f the first time it is ever generated (line 7). We can determine that a child node is being generated for the first time by comparing its parent's f with its parent's backed-up value F (line 6). If $f(n) < F(n)$ then it must have already been expanded and the child nodes must have already been generated. **[Our remark: this is similar to Observations 1 and 2 above.]** If a child has been generated previously, its F value is set to the maximum of the F value of the parent or its own f value. Propagating the backed-up values down previously explored descendants of n improves efficiency by avoiding the backtracking that would otherwise be necessary when f decreases along a path. **[Our remark: this is exactly the restore macro and the restore propagation rule].**

RBFS orders all child nodes by their current F (line 8) and expands the child with the lowest value ($best_{\mathrm{F}}$ returns the best two children according to F). **[Our remark: the fact that we only need the best two children complicates the issue.**

[9] In case of a previous collapse, $best2$ is of course the ancestor of the real $best2$ and we will need to restore it.

[10] Most of the text is copied verbatim from that paper. We thank the authors for graciously sharing the text with us.

This was maintained naturally with ILBFS as all these nodes are now added to OPEN which is ordered by definition.] Each child node with an F that is within B is visited by a recursive call **[Our remark: if it is within B it is $best$ in OPEN and will be expanded naturally by ILBFS]** using an updated threshold $min(B, F(n_2))$, the minimum cost of a node generated so far but not yet expanded. (If there is just one child then $F(n_2)$ returns ∞.) Setting the threshold this way for each recursive call allows RBFS to be robust to non-monotonic costs. **[Our remark: in ILBFS the second best node and the node with $F = B$ are nodes in OPEN. There is no need to say anything along these lines when describing ILBFS.]** If F goes down along a path, the threshold for the recursive calls will be lower than B to preserve a best-first expansion order. **[Our remark: this is preserved naturally by ILBFS which keeps a best-first order by definition]** If all child costs exceed the threshold, the search backtracks (lines 9 and 12). **[Our remark: this is the collapse macro of children up to the parent. In this case, the best node in OPEN is elsewhere in the tree.]** This is represented by B. When RBFS backtracks (line 10), it backs up either ∞ or the best f value of a node not yet expanded below node n_1, the child with the currently lowest F. Backing up a value of ∞ indicates that there are no fruitful paths below n_1 and prevents RBFS from performing any further search below it.

Comparison between ILBFS and RBFS

The most important role of ILBFS is its pedagogical structure. The author of this paper believes that it will be much easier to teach RBFS after ILBFS is first described. However, ILBFS can also be implemented on its own right and we now compare the implementation of the two algorithms.

Both algorithms will need the same amount of memory ($b \times d$) to store the principal branch and its children. While ILBFS stores all these nodes in a user supplied data structures, RBFS uses the recursion stack storing the branch and each level in the stack stores a node and its children.

As for runtime. For ILBFS the size of OPEN is $O(b \times d)$. Therefore each operation on OPEN will take $O(log(d))$ assuming that b is a constant. This time overhead is negligible. For RBFS, we note that at line 8, the algorithm returns the smallest two nodes among the children. This takes $O(b)$.[11]

In addition, it is well known that DFS-based algorithms do not need to store the entire state and when moving from a parent to a child they can only store the delta above the parent. When the DFS backtracks this delta is undone. ILBFS can implement this too because it always moves from a parent to a child or backtracks. In particular, in case 2 above, when the new best node is in another location in the search tree, ILBFS backtracks to the common ancestor via the collapse phase and then deepens to the new best node via the restore phase. Thus, these incremental changes can be implemented rather naturally in ILBFS too. Our implementation of ILBFS was around 1.3 slower than the fast implementation of RBFS which was written by Korf.

[11] In the original RBFS paper (Korf 1993) a complete sort is suggested on all children. But, since b is expected to be very small this is a negligible addition.

Therefore, from the implementing point of view, an implementer may choose either ILBFS or RBFS according to his/her personal preferences.

Conclusions

We highlighted the *collapse* and *restore* macros. We then presented ILBFS, an iterative variant of RBFS. We believe that ILBFS is easier to understand pedagogically. We thus recommend instructors to use it when they teach RBFS. Even if one still favors to implement RBFS, ILBFS is an interesting variant of RBFS and has importance on its own.

There were two problems with RBFS. First, its problematic node regenerations. This was remedied by Hatem et al. (2015). Second, it was hard to understand. Hopefully, this current paper will remedy this issue.

Acknowledgements

The work was supported by Israel Science Foundation (ISF) under grant #417/13. I thank Guni Sharon and Carlos Linarez-Lopez for valuable comments.

References

Bu, Z.; Stern, R.; Felner, A.; and Holte, R. C. 2014. A* with lookahead re-evaluated. In *SOCS*.

Edelkamp, S.; Jabbar, S.; and Lluch-Lafuente, A. 2005. Cost-algebraic heuristic search. In *AAAI, July 9-13, 2005, Pittsburgh, Pennsylvania, USA*, 1362–1367.

Felner, A.; Zahavi, U.; Holte, R.; Schaeffer, J.; Sturtevant, N. R.; and Zhang, Z. 2011. Inconsistent heuristics in theory and practice. *Artif. Intell.* 175(9-10):1570–1603.

Felner, A.; Goldenberg, M.; Sharon, G.; Stern, R.; Beja, T.; Sturtevant, N. R.; Holte, R.; and Schaeffer, J. 2012. Partial-expansion A* with selective node generation. In *AAAI*.

Goldenberg, M.; Felner, A.; Stern, R.; Sharon, G.; Sturtevant, N. R.; Holte, R. C.; and Schaeffer, J. 2014. Enhanced partial expansion A. *J. Artif. Intell. Res. (JAIR)* 50:141–187.

Hatem, M.; Kiesel, S.; and Ruml, W. 2015. Recursive best-first search with bounded overhead. In *AAAI*.

Korf, R. E. 1985. Depth-first iterative-deepening: An optimal admissible tree search. *Artif. Intell.* 27(1):97–109.

Korf, R. E. 1993. Linear-space best-first search. *Artif. Intell.* 62(1):41–78.

Neller, T. W. 2002. Iterative-refinement for action timing discretization. In *AAAI*, 492–497.

Russell, S. J. 1992. Efficient memory-bounded search methods. *Proc of ECAI-92*.

Stern, R.; Kulberis, T.; Felner, A.; and Holte, R. 2010. Using lookaheads with optimal best-first search. In *AAAI*.

Stern, R. T.; Kiesel, S.; Puzis, R.; Felner, A.; and Ruml, W. 2014. Max is more than min: Solving maximization problems with heuristic search. In *SOCS*.

Yoshizumi, T.; Miura, T.; and Ishida, T. 2000. A* with partial expansion for large branching factor problems. In *AAAI*, 923–929.