The Nature of Breadth-First Search

Article · February 1999

Source: CiteSeer

CITATIONS
11

READS
5,503

1 author:

Jason Holdsworth
James Cook University
31 PUBLICATIONS **191** CITATIONS

SEE PROFILE

The Nature of Breadth-First Search

School of Computer Science, Mathematics, and Physics
James Cook University, Australia
Technical Report 99-1

Jason J Holdsworth
Url: http://cairns.cs.jcu.edu.au/~jason

January 18, 1999

Abstract

BFS is one of the classical graph theory algorithms, typically expressed under the imperative style. However, it has applications in other domains such as artificial intelligence. All known algorithms of BFS use iteration.

This article focuses on the information maintained by BFS during exploration of an arbitrary graph component. To better understand the structure of this information, BFS is re-defined both recursively and iteratively. Also provided in this article is an analysis and comparison of the various BFS algorithms presented.

1 Introduction

Breadth-first search (BFS) is one of the oldest and most fundamental graph traversal algorithms, influencing many other graph algorithms. Early descriptions of BFS can be found in [Bel58, Lee61, Moo59]. Similarities can be found between BFS, Prim's algorithm [Pri57] for minimal spanning trees, and Dijkstra's algorithm [Dij59] for single source shortest paths. Artificial intelligence algorithms use BFS for state-space searches (see [RN95]). BFS has also been described for parallel computation [KC86], and under the functional programming paradigm [Erw92].

BFS is used to search a single connected graph component. It describes a deterministic way of exploring the vertices of the graph component. The standard definition of BFS involves an iterative loop over a queue of vertices.

Exploration of a graph component is achieved by dequeuing a vertex to discover unexplored vertices which are subsequently enqueued. This kind of manipulation produces graph exploration which can be viewed as progressive levels of discovered vertices[†].

In this article, the progressive levels of discovered vertices are interpreted in a formal way. A recurrence is defined that expresses the generative, mathematical structure realizing the progressive levels of discovered vertices (with respect to a single graph component). The recurrence is then used to construct imperative styled algorithms for BFS (recursive and iterative).

Section 2 defines the general structure (called *level sequences*) that BFS creates while exploring a graph component. Using this general structure Section 3 details recursive and iterative definitions of BFS and provides a brief analysis highlighting algorithmic differences and similarities. Section 4 ends the article with a summary and discussion.

2 The *clos* Recurrence and Level Sequences

A BFS algorithm implicitly computes the shortest path distances from a given source vertex to all other reachable vertices in a single graph component $G_c = (V_c, E_c)$. This computation is achieved by traversing a particular spanning tree rooted at the source vertex in a deterministic order. Let the spanning tree be known as the *BFS spanning tree*.

The traversal of the BFS spanning tree starts at level 0 and continues to vertices at each level of the spanning tree from level 1 to level h. A level corresponds to the distance along a path from the source vertex to a vertex in the spanning tree. Level 0 contains the source vertex. The value h is the

[†]Lee points this out and discusses its utility [Lee61].

height of the BFS spanning tree (the largest level number). Based on these concepts, a mathematical description of the traversal of the BFS spanning tree is defined.

Definition 2.1. *A sequence $\mathcal{L} = L_0 \dots L_h$ represents the traversal of a BFS spanning tree. Each $L_i = v_{i1} \dots v_{i*}$ (called a level sequence) is the sequence of vertices traversed at level i of the tree, where v_{ij} represents the j^{th} vertex traversed and v_{i*} is the last vertex traversed.*

The source vertex of the BFS spanning tree is v_{01}, and L_0 only contains v_{01}. Definition 2.1 can be visualized as the abstract tree in Figure 2.1. The distinction between each L_i indicates the separation of vertices at each level of the BFS spanning tree. Vertices in $L_0 \dots L_{h-1}$ are internal nodes of the spanning tree, and vertices in L_h are the leaf (external) nodes of the spanning tree. The concatenation of $L_0 \dots L_h$ is the overall BFS traversal sequence. The relationship between the BFS spanning tree and the level sequences can be further elaborated by the following statements:

1. All vertices in the graph component (and hence, the BFS spanning tree) are found in the union of level sequences,

$$V_c = \bigcup_{k \in 0 \dots h} L_k$$

2. Vertices and corresponding adjacent vertices in the BFS spanning tree occupy adjacent level sequences, for $i \in 0 \dots (h-1)$,

$$L_{i+1} = \{v \mid u \in L_i, (u,v) \in E_c, v \in V_c - \bigcup_{k \in 0 \dots i} L_k\}$$

3. A vertex only appears once in one of the level sequences (follows implicitly from the equation in statement 2).

Figure 2.2 shows a particular graph, and one of the possible spanning trees formed by applying a BFS algorithm. Vertices are numbered, and the spanning tree is drawn to show the level sequences. Note that each L_{i+1} is

Figure 2.1 A visualization of the structure of a BFS spanning tree.

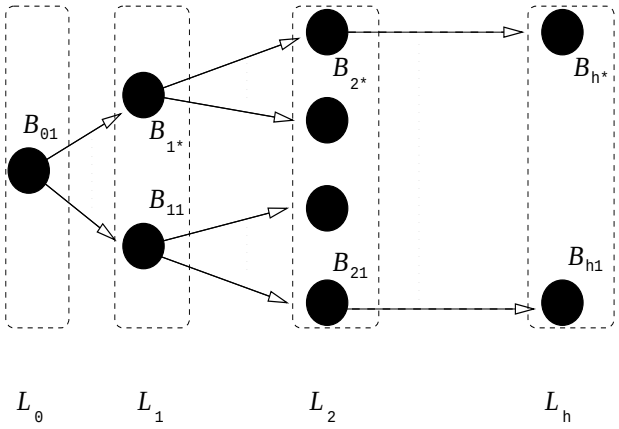
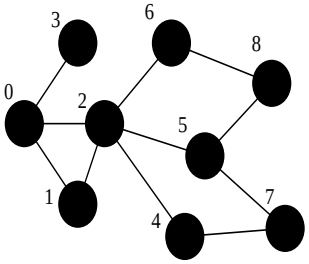
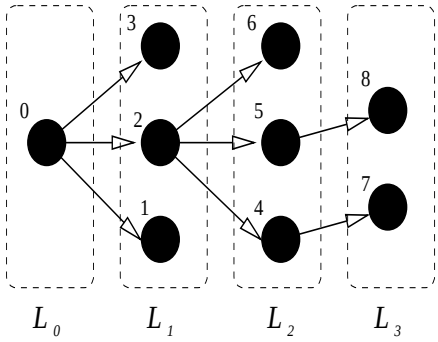


Figure 2.2 An example of level sequences.



The initial Graph.



A Spanning Tree showing closure sets.

defined based on $L_0 \ldots L_i$, where the inclusion of a vertex in L_{i+1} excludes it from future level sequences. Thus, it is possible to construct a recurrence for generating the $\mathcal{L}$ sequence.

Definition 2.2. *For $i \in 0 \ldots h$, $s \in V_c$, and $L_0 = s$, $clos(L_0)$ generates the sequence $\mathcal{L} = L_0 \ldots L_h$, where the clos recurrences is defined as,*

$$clos(L_0 \ldots L_i) \rightarrow \begin{cases} L_0 \ldots L_i & \text{if } L_{i+1} = \lambda \\ clos(L_0 \ldots L_i L_{i+1}) & \text{otherwise} \end{cases}$$

and

$$L_{i+1} = \{v \mid u \in L_i, (u,v) \in E_c, v \in V_c - \textstyle\bigcup_{k \in 0 \ldots i} L_k\}$$

Where $L_{i+1} = \lambda$ means level $(i+1)$ does not exist in the BFS spanning tree, and the calculation of L_{i+1} generates a sequence[†]. The recurrence *clos* is so named because the calculation is similar to transitive closure. Definition 2.2 leads directly to the following lemma.

Lemma 2.1. *The clos recurrence of Definition 2.2 is a valid recurrence, and $clos(L_0)$ calculates the sequence of level sequences $L_0 \ldots L_h$.*

Proof. A BFS spanning tree with h levels can be separated into the sequence $L_0 \ldots L_h$. The calculation of $clos(L_0)$ consists of h steps. The first $(h-1)$ steps occurring while $i < h$, and the final step occurring when $i = h$. The recursion discontinues when $i = h$, so the *clos* recurrence has both a recursive and base case and is therefore sound if the base case is reachable. While $i \in 0 \ldots (h-1)$ the search for $v \in V_c - \bigcup_{k \in 0 \ldots i} L_k$ is successful, since there are still vertices to be explored in the spanning tree. However, when $i = h$ it is impossible to choose v since (by statement 1) $\bigcup_{k \in 0 \ldots i} L_k = V_c$. Based on these observations, the *clos* recurrence has the following effects:

$$\begin{aligned} clos(L_0 \ldots L_i) &= clos(L_0 \ldots L_{i+1}) & i < h \\ clos(L_0 \ldots L_i) &= L_0 \ldots L_i & i = h \end{aligned}$$

Each instance of $clos(L_0 \ldots L_i)$ is replaced by $clos(L_0 \ldots L_{i+1})$ until $i = h$, at which point $clos(L_0 \ldots L_h)$ is replaced by $L_0 \ldots L_h$ (the base case is reached). Thus $clos(L_0) = L_0 \ldots L_h$. □

[†]The set construction is treated as being ordered.

Here is a brief example using the *clos* recurrence. Assume that vertices are chosen based on there number: smallest numbered vertices first. Also, assume that spaces indicate the separation between level sequences. Then the call $clos(0)$ when applied to the graph in Figure 2.2 produces the following generative calculations:

$$\begin{aligned} clos(0) &= clos(0\ 123) \\ &= clos(0\ 123\ 456) \\ &= clos(0\ 123\ 456\ 78) \\ &= 0\ 123\ 456\ 78 \end{aligned}$$

Level sequences are not obvious in the standard imperative BFS algorithm. Vertices are collected individually, with no emphasis on the distinction between level sequences. However, it is possible to define recursive and iterative algorithms to compute BFS with respect to the level sequences, and to derive the standard BFS algorithm.

3 BFS algorithms

This section describes and analyzes recursive and iterative algorithms that implement the *clos* recurrence to produce BFS. The recursive algorithm is described first, since it is a natural extension of the *clos* recurrence. Next, two iterative algorithms are described. The first iterative algorithm is the iterative version of the recursive algorithm. The second iterative algorithm is the standard BFS algorithm as a modification of the first iterative algorithm.

It is trivial to implement the *clos* recurrence using the functional paradigm (by using the *clos* recurrence directly). However, most descriptions of BFS are under the imperative paradigm. The algorithms presented here are imperative algorithms defined in an imperative notation.

Making use of the imperative global state, let G_c be global. Let ϑ be a global array of booleans associated with vertices ($\vartheta : V \rightarrow \{False, True\}$).

Before any of the algorithms are executed, assume that the boolean array is initialized ($\forall v \in V_c, \vartheta[v] \leftarrow False$). Let Adj be the global adjacency array representing E_c. BFS algorithms (such as AI algorithms) typically produce the ordered traversal of vertices as a result. The algorithms here produce an ordered set of vertices.

From these conditions, it is no longer necessary to keep track of the entire sequence $L_0 \ldots L_i$ from the *clos* recurrence, only L_i and L_{i+1} are necessary. Algorithm 3.1 calculates $clos(L_0)$ with the initial call REC$\mathrm{BFS}(L_0)$, where $L_0 = s$. The algorithmic call CONCAT is defined as follows:

$$\text{CONCAT}(a_1 \ldots a_*, b_1 \ldots b_*) \quad = a_1 \ldots a_* b_1 \ldots b_*$$

Algorithm 3.1 A recursive BFS algorithm.

REC$\mathrm{BFS}(L_i)$

```
(1)   foreach v ∈ L_i do
(2)      ϑ[v] ← True
(3)   endfor
(4)   L_{i+1} ← λ
(5)   foreach u ∈ L_i do
(6)      foreach v ∈ Adj[u] do
(7)         if ϑ[v] = False then
(8)            L_{i+1} ← CONCAT(L_{i+1}, v)
(9)         endif
(10)     endfor
(11)  endfor
(12)  if L_{i+1} ≠ λ then
(13)     return CONCAT(L_i, RECBFS(L_{i+1}))
(14)  else
(15)     return L_i
(16)  endif
```

REC$\mathrm{BFS}(L_0)$ produces the call sequence:

$$\textsc{concat}(L_0, \textsc{concat}(L_1, \ldots \textsc{concat}(L_{h-1}, L_h) \ldots))$$

This sequence is tail-recursive. After unfolding a recursive call, no further changes are made (no further recursion). For example, with $h = 3$ the call sequence is:

$$\begin{aligned}
\textsc{rec}\mathrm{BFS}(L_0) &\rightarrow \textsc{concat}(L_0, \textsc{rec}\mathrm{BFS}(L_1)) \\
&\rightarrow \textsc{concat}(L_0, \textsc{concat}(L_1, \textsc{rec}\mathrm{BFS}(L_2))) \\
&\rightarrow \textsc{concat}(L_0, \textsc{concat}(L_1, L_2)) \\
&\rightarrow \textsc{concat}(L_0, L_1 L_2) \\
&\rightarrow L_0 L_1 L_2
\end{aligned}$$

The recursive version of BFS is slightly inefficient. By definition, recursive calls must unfold to previous instances of the recursion. A number of recursive calls are made, resulting in an equal number of recursive returns. This arises because recursion is stack based (previous recursive environments are stored). However, the tail-recursive nature of BFS does not require such a history of recursive states. Thus, an iterative algorithm which passes state information forward is more desirable (state information is passed to the next instance). This requires explicit maintenance of the state information, which may drastically increase the complexity of an algorithm.

Alternatively, perhaps tail-recursion should be provided for by a syntactic construct for such situations. Recursion that 'remembers the future' is simpler and more appropriate for tail-recursive algorithms. This idea relates to concepts from the breadth-first grammars of queue automata[ACR88], and the optimization techniques employed in functional programming[FH88]. Another approach is to design compilers that detect such situations and act accordingly. Further discussion of tail-recursion is beyond the scope of this article.

Algorithm 3.2 defines the iterative counterpart to the recursive Algorithm 3.1. The state of L_i and L_{i+1} are maintained explicitly along with a state

variable (called R) which stores the overall traversal sequence. The initial call is ITERBFS(s).

Algorithm 3.2 Iterative BFS based on the recursive version.

ITERBFS(s)

```
(1)   L_i ← s
(2)   R ← s
(3)   ϑ[s] ← True
(4)   while L_i ≠ λ do
(5)      L_{i+1} ← λ
(6)      foreach u ∈ L_i do
(7)         foreach v ∈ Adj[u] do
(8)            if ϑ[v] = False then
(9)               ϑ[v] ← True
(10)              L_{i+1} ← CONCAT(L_{i+1}, v)
(11)           endif
(12)        endfor
(13)     endfor
(14)     if L_{i+1} ≠ λ then
(15)        R ← CONCAT(R, L_{i+1})
(16)     endif
(17)     L_i ← L_{i+1}
(18)  endwhile
(19)  return R
```

The sequence of calls to CONCAT for the iterative algorithm is:

$$
\begin{aligned}
R &\rightarrow L_0 \\
&\rightarrow \text{CONCAT}(R, L_1) \\
&\ldots \\
&\rightarrow \text{CONCAT}(R, L_h)
\end{aligned}
$$

The same traversal sequence is generated by the iterative algorithm and the recursive algorithm, except that the iterative algorithm is more efficient.

For example, with $h = 3$ the call sequence is:

$$\begin{aligned} R &\rightarrow L_0 \\ &\rightarrow \text{CONCAT}(L_0, L_1) \\ &\rightarrow \text{CONCAT}(L_0 L_1, L_2) \\ &\rightarrow L_0 L_1 L_2 \end{aligned}$$

In Algorithm 3.2, note that L_i and L_{i+1} have inter-dependencies. The vertices of L_i are used to generate L_{i+1}, and before subsequent iterations L_i is overwritten with the vertices in L_{i+1}. This is equivalent to utilizing a queue of vertices and appropriately ordering dequeue and enqueue operations. In fact, it is possible to simplify the iterative Algorithm 3.2 to produce the standard BFS algorithm.

The following alterations are necessary. Let Q be a queue of vertices that initially contains the source vertex. Replace the loop over $u \in L_i$ with a single dequeue operation. Replace the append operation with and enqueue operation. Relocate the assignment to R inside the loop over $v \in Adj[u]$. Single vertices are now concatenated with R rather than sequences of vertices.

Algorithm 3.3 is the result. The initial call is TRADBFS(s). The resultant concatenation sequence is:

$$\begin{aligned} R &\rightarrow v_{11} \\ &\rightarrow \text{CONCAT}(R, v_{21}) \\ &\cdots \\ &\rightarrow \text{CONCAT}(R, v_{2*}) \\ &\cdots \\ &\rightarrow \text{CONCAT}(R, v_{h1}) \\ &\cdots \\ &\rightarrow \text{CONCAT}(R, v_{h*}) \end{aligned}$$

All three algorithms compute the *clos* recurrence and thus BFS. Algorithm 3.1 is a direct interpretation of the *clos* recurrence. Algorithm 3.2 is

Algorithm 3.3 Traditional BFS as a modification of the iterative version.

TRADBFS(s)

```
(1)   Q ← {}                                   (replacing L_i & L_{i+1})
(2)   ENQUEUE(Q,s)
(3)   R ← s
(4)   ϑ[s] ← True
(5)   while not EMPTY(Q) do
(6)      u ← DEQUEUE(Q)                        (replacing the loop over L_i)
(7)      foreach v ∈ Adj[u] do
(8)         if ϑ[v] = False then
(9)            ϑ[v] ← True
(10)           ENQUEUE(Q, v)                  (replacing concat. of L_{i+1} & v)
(11)           CONCAT(R, v)                   (adjusted concat. of R & L_{i+1})
(12)        endif
(13)     endfor
(14)  endwhile
(15)  return R
```

the iterative version of Algorithm 3.1. Algorithm 3.3 is the standard BFS algorithm as a modification of Algorithm 3.2.

4 Discussion

Level sequences are a way of describing the structure of vertex ordering formed by traversing a BFS spanning tree. They can also be used to generate BFS. A recurrence for generating level sequences is the basis for recursive and iterative algorithms which compute BFS.

BFS is an important graph algorithm, and is incorporated into other graph algorithms (e.g. shortest paths [Dij59], minimal spanning trees [Pri57]). It is also useful in other disciplines, such as artificial intelligence (algorithms for searching state spaces [RN95]), and functional programming.

Recent work has focused on implementing DFS functionally [Erw97] (the method is a departure from imperative styled algorithms). BFS has so far only been described under the functional programming style in a way which is based on iteration (see [Erw92]).

The standard algorithmic technique used to formulate BFS is iteration. However, recursive BFS provides an alternative formulation that can be used in classical graph theory, artificial intelligence, and perhaps under the functional programming style. The domains of classical graph theory and artificial intelligence would use a recursive BFS algorithm similar to the definition presented in this article. Under the functional programming style, optimization techniques such as tail-recursion and possibly continuation might be used to formulate the algorithm [FH88]. Moreover, it may now be possible to formulate BFS by using the *clos* recurrence and the techniques of [Erw97].

The nature of the call-return model used in many programming paradigms causes implicit redundancy in recursive BFS and other tail-recursive algorithms. No additional recursion occurs after control is returned to previous recursive levels. A programming construct specifically for tail-recursion would be beneficial, or perhaps intelligent compilers which detect tail-recursion and act accordingly.

The definition of level sequences in this article is simplistic. BFS is only one of the two fundamental search algorithms. Depth-first search (DFS) is the other fundamental search algorithm [Tar72]. DFS constructs a different kind of spanning tree, and must remember vertices in a graph component from which there is potential exploration. The description of level sequences in this article is not sufficient for the purposes of describing the structure of DFS spanning trees. A future article will describe level sequences recursively, capturing the nature of both DFS and BFS.

For now, the current description of level sequences may prove to be a useful tool. Given information about the level sequences for a graph component, it is trivial to work out the shortest path distance between the source vertex and another vertex in the graph component: $\delta(s, v) = i$, where $v \in L_i$ and $L_i \in clos(s)$ for source vertex s. Also, a particular collection of level sequences would be the same for different spanning trees of a graph compo-

nent. Level sequences could be used to determine all such spanning trees (all re-orderings of L_i, where the ordering of vertices in L_i corresponds to the exploration of a particular spanning tree). The mathematics, algorithms, and effects arising from level sequences are a rich source for future research.

Acknowledgments

The author would like to thank Dr. Bruce Litow and Dr. Andrew Partridge for their assistance with notational conventions, programming paradigms, and numerous proof-readings.

References

[ACR88] E. Allevi, A. Cherubini, and S. Crespi Righizzi. Breadth-first phrase-structure grammars and queue automata. *Lecture Notes in Computer Science*, 324:162–170, 1988.

[Bel58] Richard E. Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958.

[Dij59] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[Erw92] M. Erwig. Graph algorithms = iteration + data structures? In *Proceedings of the 18th International Workshop on Graph-Theoretic Concepts in Computer Science*. Springer, 1992.

[Erw97] M. Erwig. Functional programming with graphs. In *Proceedings of the 2nd ACM SIGPLAN International Conference on Functional Programming (ICFP'97)*, pages 52–65, Amsterdam, June 1997.

[FH88] A. J. Field and P. G. Harrison. *Functional Programming*, pages 444–502. Addison-Wesley, 1988.

[KC86] Kim and Chwa. Parallel algorithms for a depth first search and a breadth first search. In *International Journal of Computer Mathematics*, volume 19. 1986.

[Lee61] C. Y. Lee. An algorithm for path connection and its applications. In *IRE Transactions on Electronic Computers*, volume EC-10, pages 346–365, 1961.

[Moo59] Edward F. Moore. The shortest path through a maze. In *International Symposium on the Theory of Switching*, pages 285–292. Harvard University Press, 1959.

[Pri57] R.C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36:1389–1401, 1957.

[RN95] S. Russell and P. Norvig. *Artificial Intelligence: A modern approach*, chapter 3, pages 73–87. Prentice Hall, 1995.

[Tar72] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Computing*, 1:146–160, 1972.