■ A Mathematica Tutorial for Combinatorica Users

The function below takes a graph g and a vertex start and performs a breadth first search of g starting at start. The function returns three lists (i) bfi, which contains breadth first search numbers, (ii) parent, which contains parent pointers representing the breadth first search tree, and (iii) lvl, which contains the distances of vertices from start.

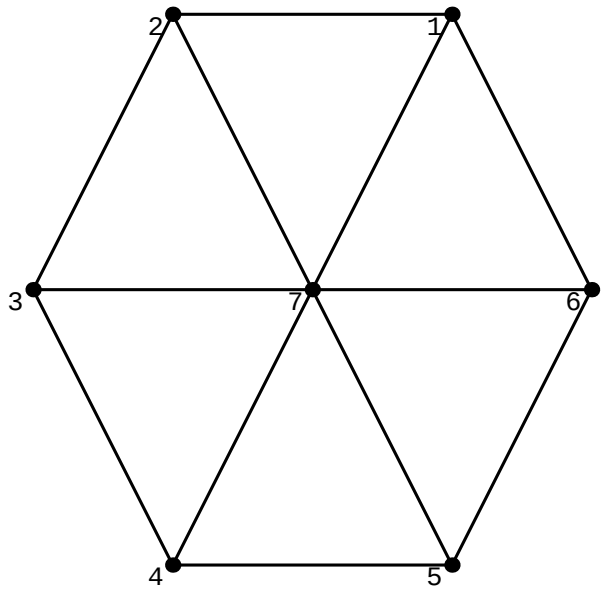
```
In[2]:= BFS[g_Graph, start_Integer] :=
         Module[{e = ToAdjacencyLists[g], bfi = Table[0, {V[g]}], cnt = 1, queue = {start},
            parent = Table[i, {i, V[g]}], lvl = Table[Infinity, {V[g]}]},
           bfi[[start]] = cnt++;
           lvl[[start]] = 0;
           While[queue ≠ {},
            {v, queue} = {First[queue], Rest[queue]};
            Scan[(If[bfi[[#]] == 0, bfi[[#]] = cnt++;
                parent[[#]] = v;
                lvl[[#]] = lvl[[v]] + 1;
                AppendTo[queue, #]]) &, e[[v]]
            ]
           ];
           {bfi, parent, lvl}] /; (1 ≤ start) && (start ≤ V[g])
```

To show how the BFS function works,we define g as a wheel with 7 vertices and run BFS on it.

```
In[3]:= g = Wheel[7]

Out[3]= Graph[12, 7, Undirected]

In[4]:= ShowGraph[g, VertexNumber → On]
```




```
Out[4]= - Graphics -

In[5]:= BFS[g, 1]

Out[5]= {{1, 2, 5, 7, 6, 3, 4}, {1, 1, 2, 7, 6, 1, 1}, {0, 1, 2, 2, 2, 1, 1}}
```

■ Function Definition

A function definition that defines a function F has the syntax

F[sequence of arguments] := body of function

Each argument in the sequence of arguments is a pattern that is matched when the function is called. For example, the first input below defines a single argument function F and the second input calls it with argument 10. The argument 10 in the function call matches x_ in the function definition with the result that x acquires the value 10 and this is used when the right handside x^2 is evaluated.

```
In[6]:= F[x_] := x^2

In[7]:= F[10]

Out[7]= 100
```

Restrictions can be placed on what matches a pattern, thereby forcing arguments of only certain types (or forms) in the call to match arguments in the definition. For example,

```
In[8]:= G[y_Integer] := y^2

In[9]:= G[10]

Out[9]= 100

In[10]:= G[10.1]

Out[10]= G[10.1]

In[11]:= F[10.1]

Out[11]= 102.01
```

In this example, only an integer argument in the function call to G can match y_Integer. This is the kind of pattern matching used in the definition of BFS. Only a graph object can match the first argument and only an integer can match the second argument. Here are examples in which the matching fails amd the function call fails to evaluate BFS.

```
In[12]:= BFS[10, 10]

Out[12]= BFS[10, 10]

In[13]:= BFS[g, 10.1]

Out[13]= BFS[Graph[12, 7, Undirected], 10.1]
```

Pattern matching is an extremely important topic inMathematicaand one can say a lot more about it than what has been said here.But,we’ll now move on to the next item in the code that is worth discussing.

■ Defining Local Variables

As the help string below tells us, a Module is a way to define local variables. The first argument of Module is a list of variables (some of which may be accompanied by initializations) and the second argument is the code to which these definitions apply.

I use Module in BFS to define the variables e, bfi, cnt, queue, parent, lvl. I initialize all of these variables as I define them.

An alternative to Module is Block. We will not discuss the distinction between these two constructs here.

```
?Module

Module[{x, y, ... }, expr] specifies that occurrences of the symbols x, y, ...  in expr should
  be treated as local. Module[{x = x0, ... }, expr] defines initial values for x, ... .
```

As the following example shows, the variables i and j have no existence outside Module.

```
In[14]:= Module[{i, j}, i = 10; j = 20]

Out[14]= 20

In[15]:= i

Out[15]= i

In[16]:= j

Out[16]= j
```

■ List Manipulation

In BFS the variable bfi is initialized as bfi=Table[0,{V[g]}]. Here I use one of the most common *Mathematica* constructs, Table.

```
In[17]:= ?Table

Table[expr, {imax}] generates a list of imax copies of expr. Table[
   expr, {i, imax}] generates a list of the values of expr when i runs from 1 to
   imax. Table[expr, {i, imin, imax}] starts with i = imin. Table[expr, {i, imin,
   imax, di}] uses steps di. Table[expr, {i, imin, imax}, {j, jmin, jmax}, ... ]
   gives a nested list. The list associated with i is outermost. Help Browser
```

As the above help strings tells us, Table can be used to generate single dimensional or multiple dimensional table (lists). For example,

```
In[18]:= Table[1, {10}]

Out[18]= {1, 1, 1, 1, 1, 1, 1, 1, 1, 1}

In[19]:= Table[1, {i, 10}]

Out[19]= {1, 1, 1, 1, 1, 1, 1, 1, 1, 1}
```

```
In[20]:= Table[i, {i, 10}]

Out[20]= {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

In[21]:= Table[i, {i, 3, 10}]

Out[21]= {3, 4, 5, 6, 7, 8, 9, 10}

In[22]:= Table[i * j, {i, 10}, {j, 10}]

Out[22]= {{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}, {2, 4, 6, 8, 10, 12, 14, 16, 18, 20},
          {3, 6, 9, 12, 15, 18, 21, 24, 27, 30}, {4, 8, 12, 16, 20, 24, 28, 32, 36, 40},
          {5, 10, 15, 20, 25, 30, 35, 40, 45, 50}, {6, 12, 18, 24, 30, 36, 42, 48, 54, 60},
          {7, 14, 21, 28, 35, 42, 49, 56, 63, 70}, {8, 16, 24, 32, 40, 48, 56, 64, 72, 80},
          {9, 18, 27, 36, 45, 54, 63, 72, 81, 90}, {10, 20, 30, 40, 50, 60, 70, 80, 90, 100}}

In[23]:= t = Table[i * j, {i, 10}, {j, 1, i}]

Out[23]= {{1}, {2, 4}, {3, 6, 9}, {4, 8, 12, 16}, {5, 10, 15, 20, 25},
          {6, 12, 18, 24, 30, 36}, {7, 14, 21, 28, 35, 42, 49}, {8, 16, 24, 32, 40, 48, 56, 64},
          {9, 18, 27, 36, 45, 54, 63, 72, 81}, {10, 20, 30, 40, 50, 60, 70, 80, 90, 100}}

In[24]:= t // ColumnForm

Out[24]= {1}
         {2, 4}
         {3, 6, 9}
         {4, 8, 12, 16}
         {5, 10, 15, 20, 25}
         {6, 12, 18, 24, 30, 36}
         {7, 14, 21, 28, 35, 42, 49}
         {8, 16, 24, 32, 40, 48, 56, 64}
         {9, 18, 27, 36, 45, 54, 63, 72, 81}
         {10, 20, 30, 40, 50, 60, 70, 80, 90, 100}
```

Though there are many other *Mathematica* constructs that can be used as alternatives to Table, they will probably be slower and not as elegant.

In BFS the variable e is initialized to the adjacency list representation of the graph g. Converting the graph into its adjacency list representation before doing anything else is common feature in many Combinatorica functions that work with graphs. Below, I show what is assigned to e when g is a 7−vertexWheel .

```
In[25]:= e = ToAdjacencyLists[g]

Out[25]= {{2, 6, 7}, {1, 3, 7}, {2, 4, 7}, {3, 5, 7}, {4, 6, 7}, {1, 5, 7}, {1, 2, 3, 4, 5, 6}}
```

The body of BFS starts after the variables are defined and the first statement in the body is

```
bfi[[ start ]] = cnt++;
```

When bfi is defined, it is initialized to a list of 0's via the Table function. In the above statement the slot in bfi indexed by start is assigned the value of cnt. So [[..]] plays the same role of as [..] does in languages such as C or C++. The second statement in BFS is a very similar initialization

```
lvl[[ start ]] = 0;
```

An arbitrary subset of the slots in a list can be assigned values using one statement, as the following example shows. After the variable l is assigned to {1, 2, ..., 10}, l[[2]], l[[5]], and l[[6]] are assigned 3 new values in one statement.

```
In[26]:= l = Range[10]

Out[26]= {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

In[27]:= l[[{2, 5, 6}]] = {-1, -2, -3}

Out[27]= {-1, -2, -3}

In[28]:= l

Out[28]= {1, -1, 3, 4, -2, -3, 7, 8, 9, 10}
```

In *Mathematica* the List object is amazingly versatile.For example,elements from a List can be extracted using any of the following operations:Part,First,Last,Head,Extract,Take,Drop,Rest,Select,and Cases. Under the heading "List Operations" *Mathematica* aprovides: Append, Prepend, Insert, Delete, DeleteCases, ReplacePart, Join, Union, Intersection, Complement, Sort, Reverse, RotateLeft, RotateRight, PadLeft, PadRight. There are many other operations related to Lists under the heading of "List Construction," "List Testing," and "Structure Manipulation." Most objects in Combinatorica are represented as lists and therefore the amazing array of list manipulation functions thatMathematicaprovides leads to compact and elegant programs.

■ Loops

The next construct used in BFS is the While construct. As the help string below tells us, the While construct takes in two arguments and the semantics are similar to those associated with while−loopsin C or C++.

```
In[29]:= ?While

        While[test, body] evaluates test, then body, repetitively, until test first fails to give True.
         Help Browser
```

The example below shows that unlike a function like Table, While does not produce any output and therefore its importance is in the side effects it causes.

```
In[38]:= sum = 0

Out[38]= 0

In[39]:= i = 0

Out[39]= 0

In[40]:= While[i < 10, sum = sum + 10; i++]

In[41]:= sum

Out[41]= 100
```

Other iteration constructs in *Mathematica* are provided by the Do−constructand the For−construct.

```
In[42]:= ?Do

        Do[expr, {imax}] evaluates expr imax times. Do[expr, {i, imax}] evaluates
           expr with the variable i successively taking on the values 1 through imax (in
           steps of 1). Do[expr, {i, imin, imax}] starts with i = imin. Do[expr, {i, imin,
           imax, di}] uses steps di. Do[expr, {i, imin, imax}, {j, jmin, jmax}, ... ]
           evaluates expr looping over different values of j, etc. for each i. Help Browser
```

```
In[43]:= ?For

        For[start, test, incr, body] executes start, then repeatedly
           evaluates body and incr until test fails to give True. Help Browser
```

The first statement in the body of the while−loopin BFS is

```
{ v, queue } = { First[queue], Rest[queue] };
```

and this is yet another evidence of the versatility of the List data structure in *Mathematica*. First[queue] evaluates to the first (top) element in the queue. Rest[queue] evaluates to a list obtained by dropping the first element from queue. These two quantities are simultaneously assigned to v and queue respectively, having the effect of dequeuing an element from queue and assigning it to v.

```
In[44]:= l = {1, 3, 8, -1, 11, 12}

Out[44]= {1, 3, 8, -1, 11, 12}

In[45]:= {v, l} = {First[l], Rest[l]}

Out[45]= {1, {3, 8, -1, 11, 12}}

In[52]:= {v, l}

Out[52]= {1, {3, 8, -1, 11, 12}}
```

■ Functional Operations

The rest of the while−loopin BFS is taken up by a call to the Scan function. Scan, along with relatives such as Map and Apply and extremely important functions in *Mathematica*. These are called *functional operations* because they work by applying functions to elements of lists.

```
In[53]:= ?Scan

        Scan[f, expr] evaluates f applied to each element of expr in turn. Scan[f, expr,
           levelspec] applies f to parts of expr specified by levelspec. Help Browser
```

Scan, like the While−functionproduces no output. It is useful only for the side−effectsit produces. In BFS, Scan produces side−effectson bvi, parent, and lvl.

In the following example, I define a single argument function Acc that increases a quantity m by the given argument. Scan[Acc, l] evaluates Acc by providing it with each element of l in turn. This results in each element in l being added to m. If m is initialized to 0 and then Scan[Acc, l] is called then m ends up being the sum of the values in l. Note that Scan[Acc, l] produces no output and the this function call is useful only in the effect it has on m.

```
In[84]:= Acc[x_] := m = m + x

In[85]:= l

Out[85]= {3, 8, -1, 11, 12}

In[86]:= m = 0

Out[86]= 0

In[87]:= Scan[Acc, l]
```

```
In[88]:= m

Out[88]= 33
```

In the above example, Acc was explicitly named. *Mathematica* provides a way of specifying functions without explicitly naming them. This mechanism, demonstrated below is called a "pure function." In the example below, we do not use Acc; instead, within the call to Scan, we specify an equivalent pure function

Function[x, m = m + x]

that takes a single formal argument x and adds it to the quantity m.

```
In[89]:= m = 0

Out[89]= 0

In[90]:= Scan[Function[x, m = m + x], l]

In[91]:= m

Out[91]= 33

In[92]:= ?Function

        Function[body] or body& is a pure function. The formal parameters are # (or #1), #2, etc.
           Function[x, body] is a pure function with a single formal parameter x. Function[{x1,
           x2, ... }, body] is a pure function with a list of formal parameters. Help Browser
```

The help string for Function above tells us that there is a shortform we can use for pure functions in which the formal arguments are also not explicitly named. The example below shows this. In this example,

(m = m + #)&

is used to specify a pure function with a single argument, specified by #.

```
In[105]:= m = 0

Out[105]= 0

In[106]:= Scan[(m = m + #) &, l]

In[107]:= m

Out[107]= 33
```

Below is yet another example of Scan and pure functions.The pure function

If [PrimeQ[#], s++]&

increments s if the argument provided to the function is a prime. Using this, the following example discovers that the number of primes no greater than 1000 is 168.

```
In[108]:= l = Range[1000];

In[109]:= ?PrimeQ

        PrimeQ[expr] yields True if expr is a prime number, and yields False otherwise. Help Browser

In[110]:= s = 0

Out[110]= 0

In[111]:= Scan[If[PrimeQ[#], s++] &, l]
```

```
In[112]:= s

Out[112]= 168
```

As a final example of Scan, consider the function call to Scan in BFS. The appropriate code fragment is reproduced below. The second argument to this function call is e[[v]], which is the list of all neighbors of v. So Scan processes each neighbor, say u, of v in turn, checking to see the bfi−valueof u is 0. If bfi[[u]] is then it means that u has not yet been visited and the values of bfi[[u]], parent[[u]], and lvl[[u]] are all appropriately updated.

```
 Scan[(If[bfi[[#]]==0,bfi[[#]]=cnt++;
parent[[#]]=v;
        lvl[[#]]=lvl[[v]]+1;
AppendTo[queue,#]])&,    e[[v]]
]
```

Map is a Mathematica function that is closely related to Scan and is probably more useful than Scan. The big difference between Scan and Map is that Map returns output and therefore can be used as an argument to other functions. Anything Scan can do, Map can also do. But calling Map when Scan suffices, as in the BFS example leads to inefficient code.

```
In[113]:= ?Map

        Map[f, expr] or f /@ expr applies f to each element on the first level in expr. Map[f,
           expr, levelspec] applies f to parts of expr specified by levelspec. Help Browser

In[114]:= l = {1, 2, 10, 4}

Out[114]= {1, 2, 10, 4}

In[115]:= Map[#^2 &, l]

Out[115]= {1, 4, 100, 16}

In[116]:= Map[f, {1, 3, 4}]

Out[116]= {f[1], f[3], f[4]}
```

Map is important enough that *Mathematica* contains several variants: MapAll, MapAt, MapIndexed, and MapThread.

■ Final Words

After the end of the definition of BFS, the reader will notice

/; (1 ≤ start) && (start ≤ V[g])

As the following help string tells us, this is a boolean condition that has to evaluate to True for the function definition to help. So this condition ensures that unless start is a valid vertex (an integer in the range 1 through the number of vertices) BFS will not be defined.

```
In[117]:= ?/;

        patt /; test is a pattern which matches only if the evaluation of test yields True. lhs :>
           rhs /; test represents a rule which applies only if the evaluation of test yields True.
           lhs := rhs /; test is a definition to be used only if test yields True. Help Browser
```