Breadth First Search Applications

- Using the template method pattern, we can specialize the BFS traversal of a graph G to solve the following problems in $O(n + m)$ time
  - Compute the connected components of G
  - Compute a spanning forest of G
  - Find a simple cycle in G, or report that G is a forest
  - Given two vertices of G, find a path in G between them with the minimum number of edges, or report that no such path exists

BFS Algorithm (revised)

- The BFS algorithm using a single sequence L

```
Algorithm BFS(G) {top level}
    Input graph G
    Output labeling of the edges
           and partition of the
           vertices of G
  for all u ∈ G.vertices()
    setLabel(u, UNEXPLORED)
  for all e ∈ G.edges()
    setLabel(e, UNEXPLORED)
  for all v ∈ G.vertices()
    if getLabel(v) = UNEXPLORED
      BFS(G, v)
```

```
Algorithm BFS(G, s)
  L ← new empty sequence
  L.insertLast(s)
  setLabel(s, VISITED)
  while ¬L.isEmpty()
    v ← L.removeAtRank(0)
    for all e ∈ G.incidentEdges(v)
      if getLabel(e) = UNEXPLORED
        w ← opposite(v,e)
        if getLabel(w) = UNEXPLORED
          setLabel(e, DISCOVERY)
          setLabel(w, VISITED)
          L.insertLast(w)
        else
          setLabel(e, CROSS)
```


Template Version of BFS

```
Algorithm BFS(G) {top level}
    Input graph G
    Output labeling of the edges of
           G as discovery edges and
           cross edges
  initResult(G)
  for all u ∈ G.vertices() do
    setLabel(u, UNEXPLORED)
    initVertices(u)
  for all e ∈ G.edges() do
    setLabel(e, UNEXPLORED)
    initEdges(e)
  for all v ∈ G.vertices() do
    if getLabel(v) = UNEXPLORED
      preComponentVisit(G, v)
      BFS(G, v)
      postComponentVisit(G, v)

  result(G)
```

```
Algorithm BFS(G, s)
    setLabel(s, VISITED)
    L.insertLast(s)
    startBFS(G, s)
    while ¬L.isEmpty() do
      v ← L.removeAtRank(0)
      preVertexVisit(G, v)
      for all e ∈ G.incidentEdges(v) do
        if getLabel(e) = UNEXPLORED
          w ← opposite(v, e)
          preEdgeVisit(G, v, e, w)
          if getLabel(w) = UNEXPLORED
            preDiscEdgeVisit(G, v, e, w)
            setLabel(e, DISCOVERY)
            setLabel(w, VISITED)
            L.insertLast(w)
            postDiscEdgeVisit(G, v, e, w)
          else
            setLabel(e, CROSS)
            crossEdgeVisit(G, v, e, w)
      postVertexVisit(G, v)

    finishBFS(G, s)
```


Finding a Simple Path

Overriding template methods in subclass FindSimplePath

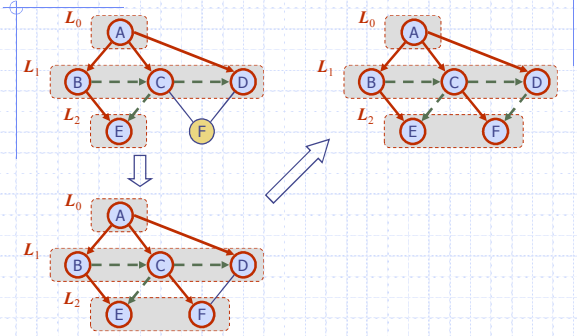
```
Algorithm preDiscEdgeVisit(v, e, w)
    setParent(w, v)     {v is w's parent}
    setEdge(w, e)       {e is the edge to w's parent v}
    if w=z then         {z is the target vertex}
      S ← new empty stack   {could use a Q or sequence instead}
      createAndSavePath( w, S )
      path ← S

Algorithm createAndSavePath(w, S)
    v ← getParent(w)
    if v = null then
      S.push(w)                 {push current vertex w}
    else
      S.push(w)                 {push current vertex w}
      S.push(getEdge(w))        {push edge connecting w to v}
      createAndSavePath(v, S)
```


- What is missing?
- Top level method/function

```
Algorithm findPath(G, v, w)
  for all u ∈ G.vertices() do
    setLabel(u, UNEXPLORED)
  for all e ∈ G.edges() do
    setLabel(e, UNEXPLORED)
  z ← w
  BFS(G, v)
```

BFS Example



Finding a Simple Cycle

Overriding template methods in subclass FindCycle

Algorithm ***preDiscEdgeVisit(G, v, e, w)***
setParent(w, v) {v is w's parent}
setEdge(w, e) {e is edge to w's parent}

Algorithm ***crossEdgeVisit(G, v, e, w)***
S1 ← new empty stack
createAndSavePath(v, S1)
S2 ← new empty stack
createAndSavePath(w, S2)
C ← **createAndSaveCycle**(S1 , S2, e)
result.insertLast(C)

Does this work?
What should createAndSaveCycle do?

Reconstructing the cycle

Algorithm ***createAndSaveCycle(S1, S2, e)***
while ¬ ***S1***.isEmpty( ) /\ ¬ ***S2***.isEmpty( ) /\ ***S1***.top( ) = ***S2***.top( ) **do**
p1 ← ***S1***.pop( ) {eliminate the path preceding the cycle}
S2.pop( )

C ← new empty sequence
C.insertFirst(p1) {insert the vertex connecting the two paths}
while ¬ ***S2***.isEmpty( ) **do**
C.insertLast(***S2***.pop( )) {insert path in S2 at end of C}

C.insertLast(e) {insert cross edge at end of C}

while ¬ ***S1***.isEmpty( ) **do**
C.insertFirst(***S1***.pop( )) {insert path in S1 at front of C}

return C