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

- The BFS algorithm using a single sequence/list/queue 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
        BFScomponent(G, v)
```

```
Algorithm BFScomponent(G, s)
  setLabel(s, VISITED)
  L ← new empty List
  L.insertLast(s)
  while ¬L.isEmpty() do
      v ← L.remove(L.first())
      for all e ∈ G.incidentEdges(v) do
        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)
      postInitVertex(u)
  for all e ∈ G.edges() do
      setLabel(e, UNEXPLORED)
      postInitEdge(e)
  for all v ∈ G.vertices() do
      if isNextComponent(G, v)
         preComponentVisit(G, v)
         BFScomponent(G, v)
         postComponentVisit(G, v)

  return result(G)
```

Algorithm ***isNextComponent***(***G, v***)
```
  return getLabel(v) = UNEXPLORED
```

Algorithm ***BFScomponent***(***G, s***)
```
  setLabel(s, VISITED)
  L ← new empty List
  L.insertLast(s)
  startBFScomponent(G, s)
  while ¬L.isEmpty() do
     v ← L.remove(L.first())
     preVertexVisit(G, v)
     for all e ∈ G.incidentEdges(v) do
        preEdgeVisit(G, v, e, w)
        if getLabel(e) = UNEXPLORED
           w ← opposite(v, e)
           unexploredEdgeVisit(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=target  then       {target is the target vertex (subclass field)}
            S ← new empty Stack     {could use a Q or Sequence instead}
            createAndSavePath( w, S )
            path ← S

Algorithm result(G)
        return  path
Algorithm startBFS(G, s)
        setParent(s, ∅)      {starting vertex has null parent}

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

- What is missing?
  - Top level method/function

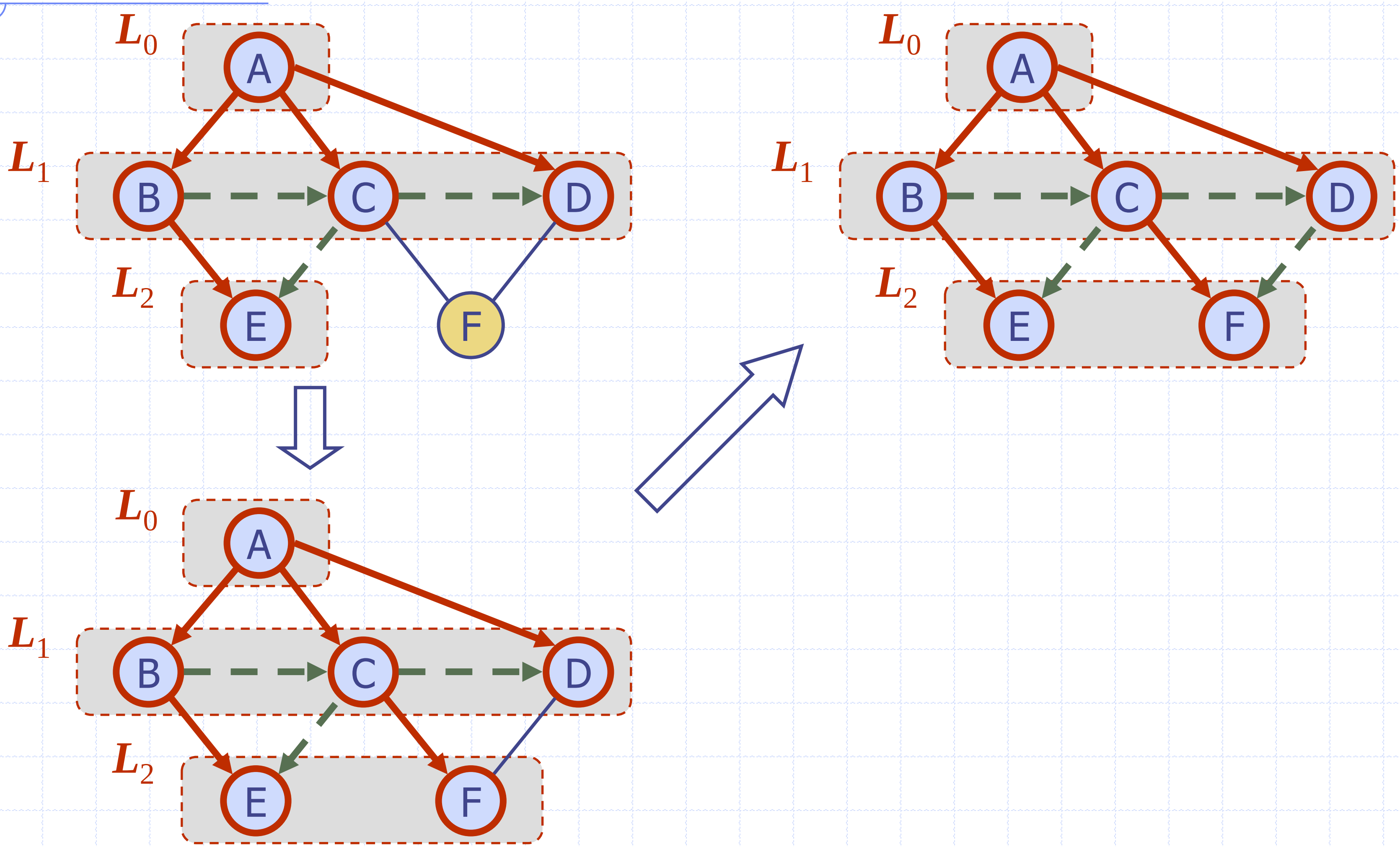
```
Algorithm findPath(G, v, w)
  for all u ∈ G.vertices() do
     setLabel(u, UNEXPLORED)
  for all e ∈ G.edges() do
     setLabel(e, UNEXPLORED)
  setParent(v, ∅)     {v has null parent}
  target ← w    {subclass field}
  BFScomponent(G, v)
  return path
```

New and Better FindSimplePath

Better FindSimplePath

```
Algorithm findSimplePath(G, v, w)        {Top Level Function and Interface}
        startV ← v               {subclass field}
        endV ← w                 {subclass field}
        return  BFS(G)

Algorithm isNextComponent(G, v) {so we start on correct vertex}
     return  v = startV            {and don't have to initialize vertices and edges}

Algorithm startBFS(G, s)
   setParent(s, ∅)     {s has null parent indicating starting vertex}

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}

Algorithm result(G)
       S ← new empty Stack       {use a Stack this time, reverses order}
       createAndSavePath(endV, S)
       return S

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

BFS Example



Finding a Simple Cycle

Overriding template methods in subclass FindCycle

Algorithm ***findCycle***(*G*)
    return BFS(G)

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 from the two paths

```
Algorithm createAndSaveCycle(S1, S2, e)
    while ¬ S1.isEmpty( ) /\ ¬ S2.isEmpty( ) /\ S1.top( ) = S2.top( )  do
        last ← S1.pop( )   {need to save vertex connecting the two paths}
        S2.pop( )    {eliminate path preceding the cycle}

    C  ← new empty Sequence
    C.insertFirst(last)    {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
        last ← S1.top( )   {save previous element of cycle}
        C.insertFirst(S1.pop( ))  {insert path in S1 at front of C}

    C.insertLast(last)    {insert the same vertex at both ends of the cycle}
    return  C
```

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)

  return 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)
```

Overriding methods in Subclass FindCycle (another version)

Algorithm ***findCycle***(*G*)
    return BFS(G)

Algorithm ***preComponentVisit***(*G*, *v*)
    setLevel(v, 0) {set level of first vertex v to 0}

Algorithm ***preDiscEdgeVisit***(*G*, *v*, *e*, *w*)
    setLevel(w, getLevel(v) + 1) {set level of w}
    setParent(w, e) {e is edge to w's parent}

How should we create the Cycle?

(inspired by Shivali Jain, July 2017)

Constructing the cycle from Parent (edge to parent) attribute

```
Algorithm crossEdgeVisit (G, v, e, w)
    C  ← new empty List (or Sequence)
    C.insertLast(e)   {insert into C the cross edge that connects two paths}

    if getLevel(v) > getLevel(w)  then
        C.insertFirst(v)   {insert v before e in cycle since path to v is longer}
        C.insertFirst(getParent(v))   {insert edge to parent before v in cycle}
        v ← G.opposite(getParent(v) ,v)  {even up the path lengths}
    else if getLevel(v) <  getLevel(w)  then
        C.insertLast(w)   {insert w after e in cycle since path to w is longer}
        C.insertLast(getParent(w))   {insert edge to parent after w in cycle}
        w ← G.opposite(getParent(w),w)  {even up the path lengths}

    while  v ≠ w do
        C.insertFirst(v)   {insert v at front of cycle}
        C.insertFirst(getParent(v))   {insert edge to parent at front also}
        v ← G.opposite(getParent(v), v)  {move to the next vertex in path}
        C.insertLast(w)   {insert w at end of cycle}
        C.insertLast(getParent(w))   {insert edge to parent at end also}
        w ← G.opposite(getParent(w), w)  {move to the next vertex in path}

    C.insertFirst(v)   {insert v at front of cycle; two paths meet at v to form cycle}
    C.insertLast(v)   {insert v at end of cycle }

    result.insertLast( C )
```