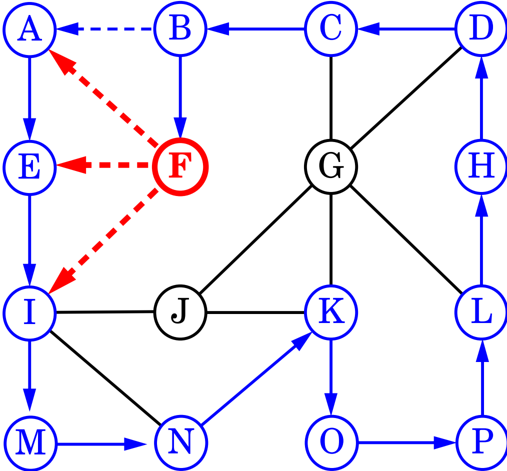
GRAF-TRAVERSERING

- Dybde-Først Søk
- Bredde-Først Søk
- Bruk av 'MetodeMal' som designmønster (Template Method Pattern

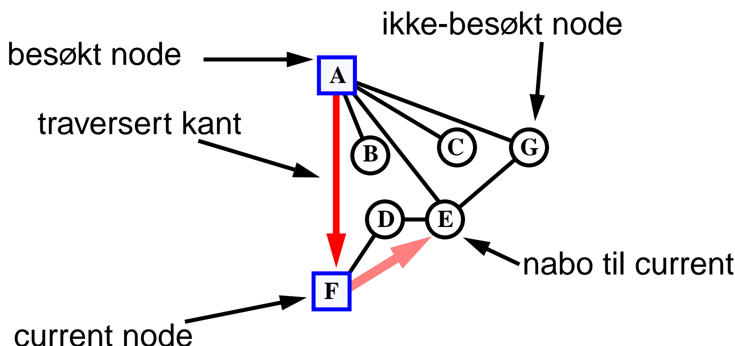


Hvordan utforske en labyrint uten å gå seg vill.

- Et **dybde-først søk (DFS)** i en urettet graf G er som å vandre gjennom en labyrint med tau og rødmaling uten å gå seg vill.
- Vi starter i node *s*, knytter tauet til et fast punkt og maler noden *s* rød. La *s* være current node *u*.
- Gå så, med tauet, langs en vilkårlig kant (*u*,*v*).
- Om (*u*,*v*) leder oss til en allerede rødmalt node *v* så nøst opp tauet og gå tilbake til *u*.
- Om *v* derimot ikke er rød, så maler vi den rød, setter *v* til å være den nye *u* og fortsetter som over.
- Tilslutt vil alle kanter fra *u* lede til rødmalte noder. Vi følger da tauet tilbake (dette er **backtracking**) og nøster tauet opp til den foregående noden i tauet. Dette er den røde noden *v* som tok oss hit. Denne *v* blir nå den nye *u* og vi fortsetter som over.
- Når vi backtracker helt til startnoden *s* og alle kanter utfra *s* leder til rødmalte noder, så er vår DFS-utforskning avsluttet.

Dybde-Først Søk

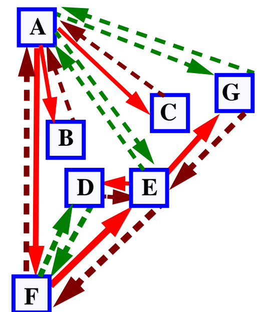
Algorithm DFS(*u*);
Input: En node *u* i graf G
Output: Merking av kanter som enten "discovery"-kanter eller "tilbake"-kanter
merk u som besøkt
for hver kant *e=(u,v)* incident med node u **do**
  if *v* ikke besøkt **then**
    merk *e* som discovery-kant
    recursivt kall til **DFS**(*v*)
  else
    merk *e* som tilbake-kant



Rekkefølge på kanter:

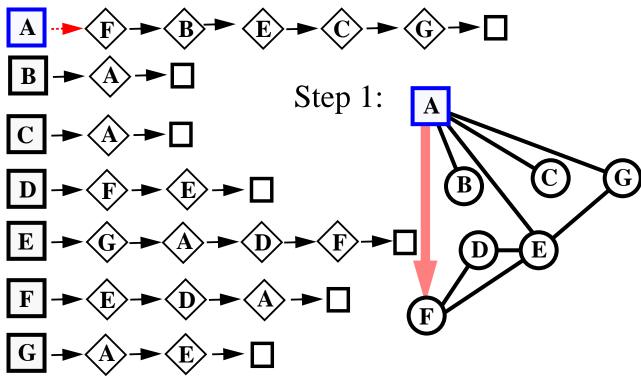
- DFS avhenger av gitt rekkefølge på kanter.
- Om vi starter i A og ser først på kanten til F, så til B, så E, C, og tilslutt G

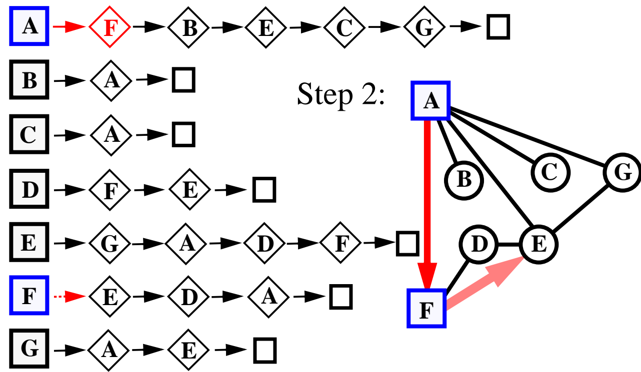
- Resultatet blir:
  discovery-kant
  tilbake-kant
  backtrack

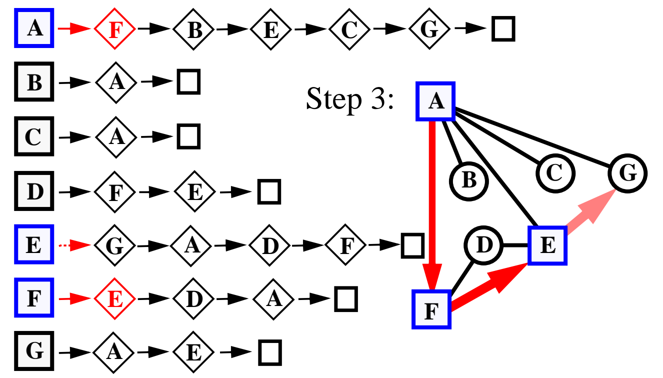


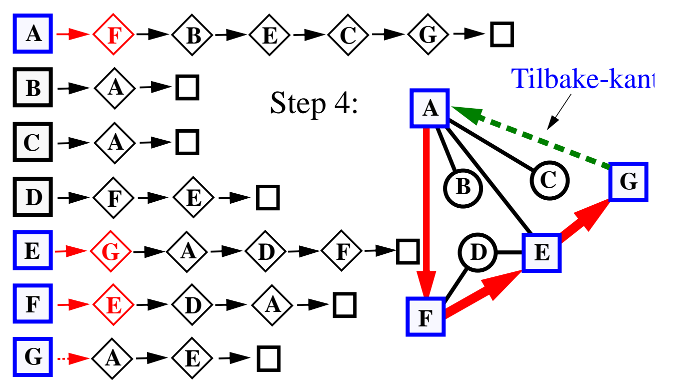
Derimot, om vi starter i A og ser først på F, som før, men deretter C, så E, B, og tilslutt F,

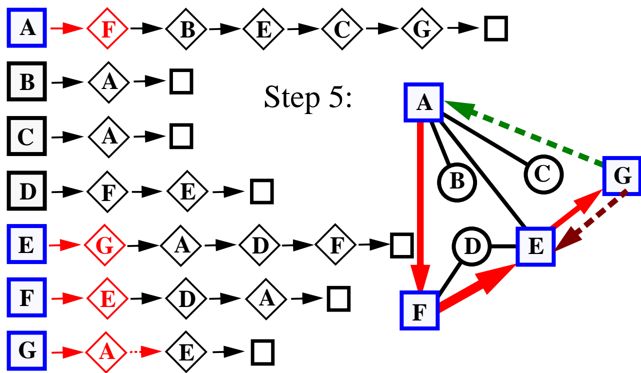
- så blir backEdges, discoveryEdges forskjellige.

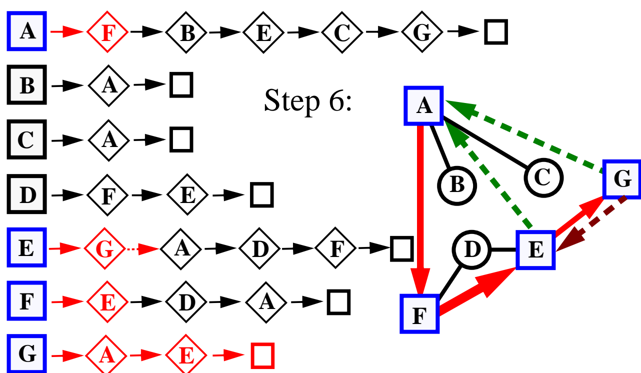

Step 1:

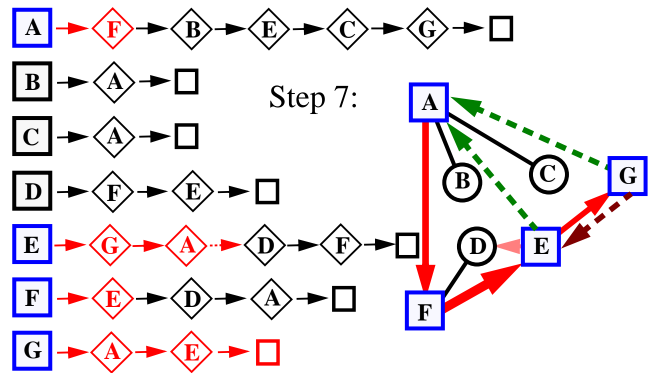

Step 2:

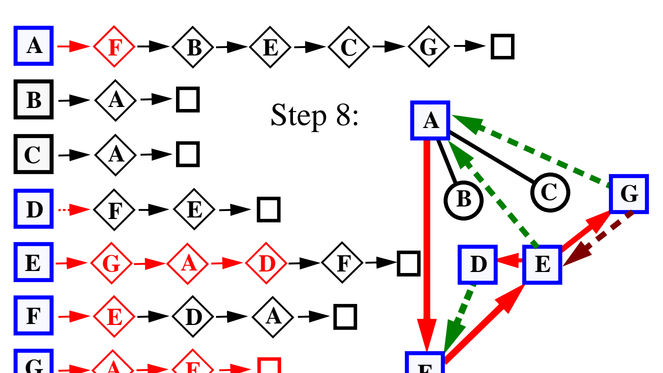

Step 3:

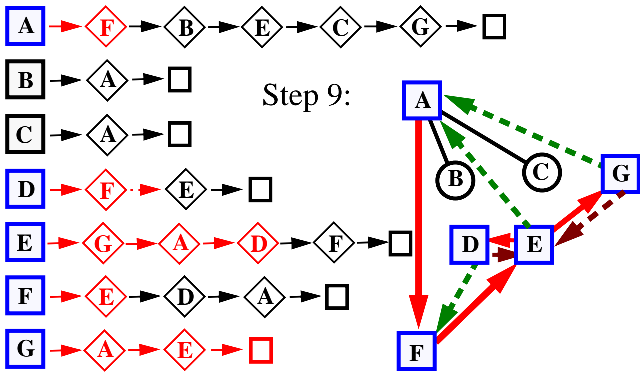

Step 4:
Tilbake-kant

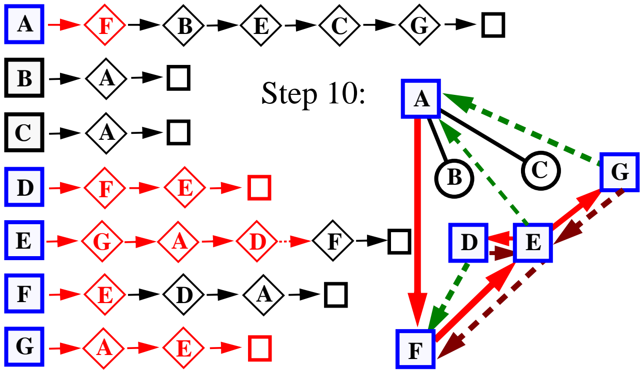

Step 5:

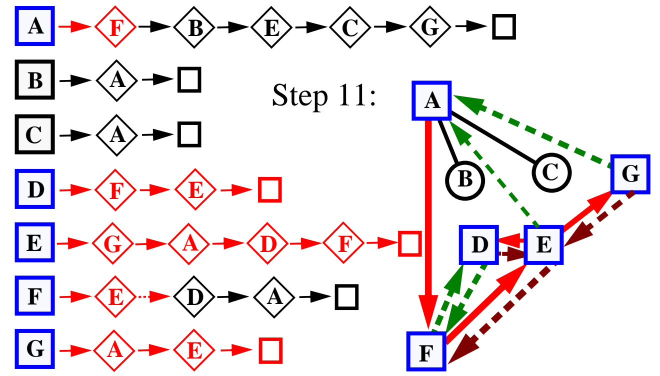

Step 6:

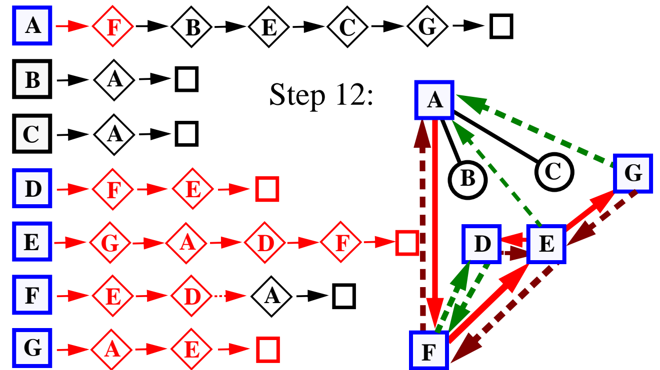

Step 7:

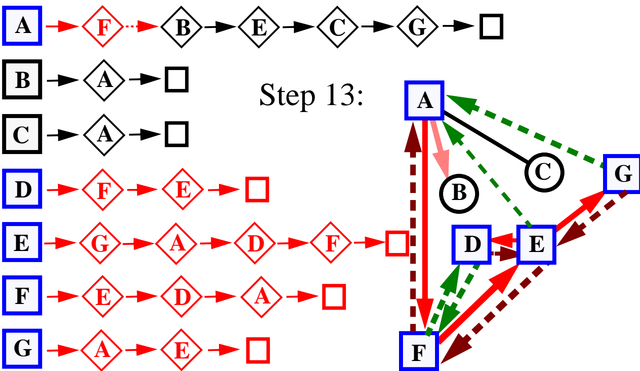

Step 8:

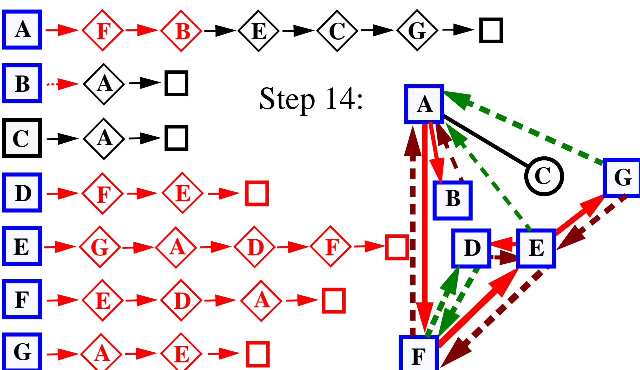

Step 9:
A F B E C G
B A
C A
D F E
E G A D F
F E D A
G A E

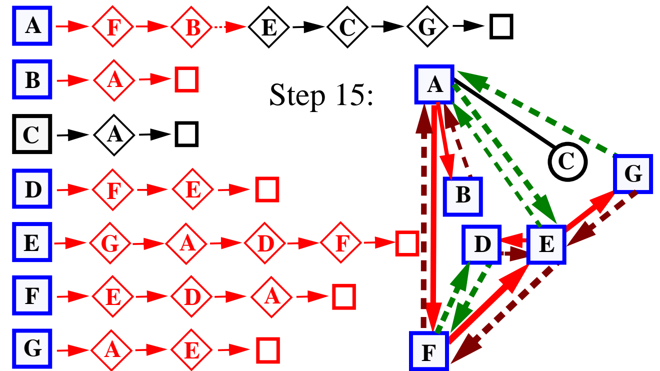

Step 10:
A F B E C G
B A
C A
D F E
E G A D F
F E D A
G A E

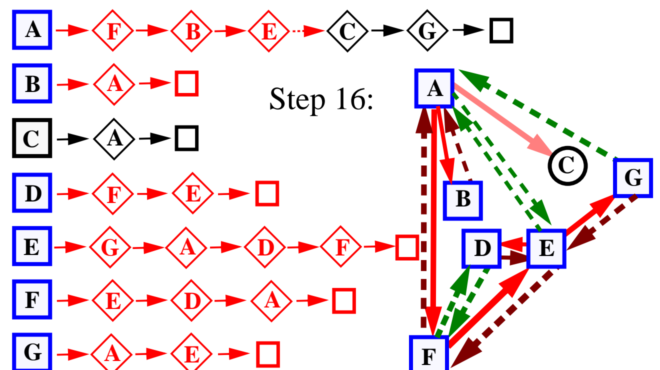

Step 11:
A F B E C G
B A
C A
D F E
E G A D F
F E D A
G A E

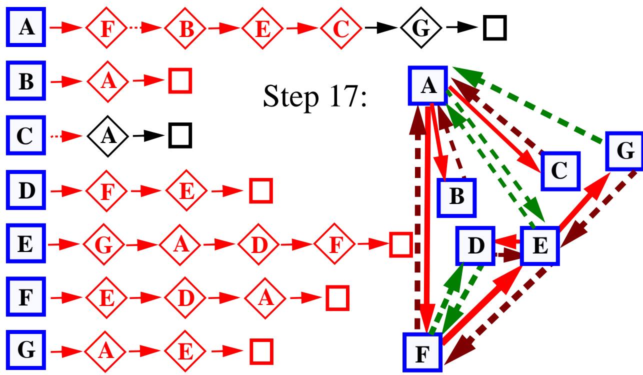

Step 12:
A F B E C G
B A
C A
D F E
E G A D F
F E D A
G A E

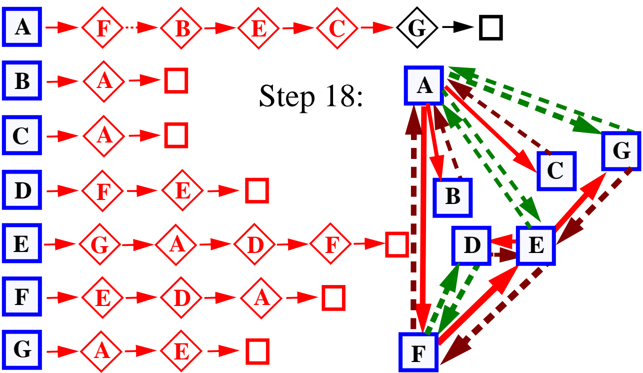

Step 13:
A F B E C G
B A
C A
D F E
E G A D F
F E D A
G A E

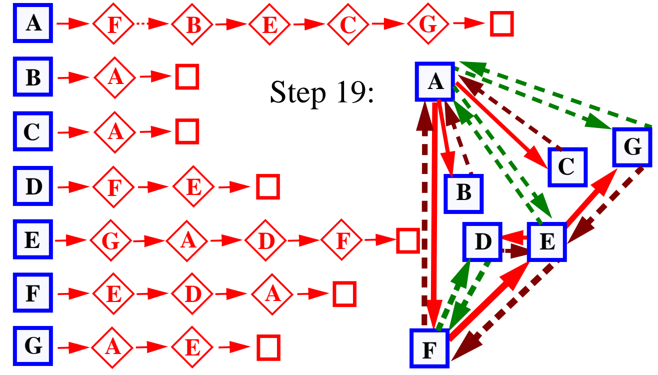

Step 14:
A F B E C G
B A
C A
D F E
E G A D F
F E D A
G A E


Step 15:
A F B E C G
B A
C A
D F E
E G A D F
F E D A
G A E


Step 16:
A F B E C G
B A
C A
D F E
E G A D F
F E D A
G A E







Presto!

DFS-Egenskaper

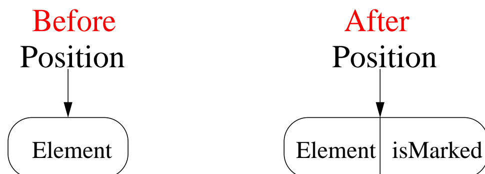
- Proposisjon 9.12 i læreboka: 'La G være urettet graf hvor vi kjører DFS med startnode *s*. Da har vi at
  - 1) alle noder i den sammenhengende komponent til *s* ble besøkt under DFS, og
  - 2) discovery-kantene utgjør et utspennende tre i den sammenhengende komponenten til *s*'
- Bevis for 1):
  - Bevis ved motsigelse. Anta en node w ikke ble besøkt. Det finnes en sti fra s til w. La v være første node på denne stien som ikke ble besøkt (kan f.eks ha v=w). Dvs noden foran v på stien, kall denne u, ble faktisk besøkt. Når vi besøkte u må vi ha sett på kanten (u, v), og da ville vi ha besøkt v. Dette motsier antagelsen.
- Bevis for 2):
  - Vi merker en kant (u,v) som 'discovery' kun når den går fra besøkt node u til ikke-besøkt node v. Dvs vi skaper ingen sykel av discovery-kanter. Siden alle noder tilslutt er besøkt må disse kantene utgjøre et utspennende tre.

Kjøretids-Analyse: med riktig graf-representasjon er DFS en lineær-tids algoritme.

- Husk:
  - DFS blir kallt rekursivt for hver node nøyaktig en gang.
  - Hver kant e=(u,v) utforskes to ganger, når vi går gjennom naboene til u og når vi går gjennom naboene til v.
  - Anta komponenten til startnoden s har n_s noder og m_s kanter. DFS fra *s* tar da O(n_s +m_s) tid om:
  - Datastruktur som brukes for grafen gir konstant-tid O(1) metoder for å aksessere hver nabo,
  - Merking av noder og kanter og sjekking av denne merking tar O(1) tid,
  - Siden vi merker noder, kan vi raskt oppdage om en kant leder til ikke-besøkt node.

Merking av noder

- Hvordan merke noder?.
- Utvid node-posisjoner med en boolsk variabel T=Merket og F=Ikke-merket



- Bruk hash-tabell (mer om dette senere i kurset)

Bruk av MetodeMal : The Template Method Pattern

- **MetodeMal** er et programmeringsdesignmønster som bruker en ***generisk algoritme*** som mal for flere spesialiseringer til bestemte applikasjoner.
- Vi designer en 'abstract class MalX' som
  - implementerer kun ***skjelettet*** til en algoritme
  - den klassen bruker flere 'abstract method''s
  - disse metodene gies en implementasjon i en konkret 'class SpesialX extends MalX'.
- Dette forenkler gjenbruk av kode på en ryddig og fornuftig måte.
- Eksempler
  - ***generisk traversering av binært tre*** (som preorden, inorden, og postorden) og applikasjoner av dette
  - ***generisk dybde-først søk i en urettet graf*** og applikasjoner av dette

Generisk DFS

```
public abstract class DFSMAL {
protected InspectableGraph graph;
protected Object visitResult;
protected Object dfsVisit(Vertex v) {
initResult();
   startVisit(v);
   mark(v);
   for (Enumeration inEdges = graph.incidentEdges(v);
        inEdges.hasMoreElements();) {
    Edge nextEdge = (Edge) inEdges.nextElement();
    if (!isMarked(nextEdge)) { // found an unexplored edge
     mark(nextEdge);
     Vertex w = graph.opposite(v, nextEdge);
     if (!isMarked(w)) { // discovery edge
       mark(nextEdge);
       traverseDiscovery(nextEdge, v);
       if (!isDone())
         visitResult = dfsVisit(w); }
     else  // back edge
      traverseBack(nextEdge, v);
    }
   }
   finishVisit(v);
   return result();
 }
```


Støttemetoder for generisk DFS

```
public abstract Object execute (InspectableGraph g,
Vertex start, Object info);

protected abstract void initResult();

protected abstract void startVisit(Vertex v);

 protected abstract void traverseDiscovery(Edge e,
Vertex from);

protected abstract void traverseBack(Edge e, Vertex
from);

protected abstract boolean isDone();

protected abstract void finishVisit(Vertex v);

protected Object result() { return new Object(); }
```

1.Spesialisering av generisk DFS

- class FindPath spesialiserer DFSMAL til å finne en sti fra noden start til noden target.
  public class FindPathDFS extends DFSMAL {

```
protected Sequence path;
protected boolean done;
protected Vertex target;
public Object execute(InspectableGraph g, Vertex start,
                      Object info) {
  super.execute(g, start, info);
  path = new NodeSequence();
  done = false;
  target = (Vertex) info;
  dfsVisit(start);
  return path.elements();
}
protected void startVisit(Vertex v) {
  path.insertFirst(v);
  if (v == target) { done = true; }
}
protected void finishVisit(Vertex v) {
  if (!done) path.remove(path.first());
}
protected boolean isDone() { return done; }
}
```


2.Spesialisering av generisk DFS

- FindAllVertices spesialiserer DFSMAL til å returnere en enumeration av alle noder i sammenhengende komponent til start -noden.

```
public class FindAllVerticesDFS extends DFSMAL {
  protected Sequence vertices;
  public Object execute(InspectableGraph g, Vertex start,
                        Object info) {
    super.execute(g, start, info);
    vertices = new NodeSequence();
    dfsVisit(start);
    return vertices.elements();
  }

  public void startVisit(Vertex v) {
    vertices.insertLast(v);
  }
}
```


3.Spesialisering av generisk DFS

- ConnectivityTest bruker en spesialisert DFS for å teste om en graf er sammenhengende.

```
public class ConnectivityTest {
  protected static DFS tester = new
FindAllVerticesDFS();
  public static boolean isConnected(InspectableGraph g)
  {
    if (g.numVertices() == 0) return true; //empty is
                                            //connected
    Vertex start = (Vertex)g.vertices().nextElement();
    Enumeration compVerts =
             (Enumeration)tester.execute(g, start, null);
    // count how many elements are in the enumeration
    int count = 0;
    while (compVerts.hasMoreElements()) {
      compVerts.nextElement();
      count++;
    }
    if (count == g.numVertices()) return true;
    return false;
  }
}
```


4.Spesialisering av generisk DFS

- FindCycle spesialiserer DFSMAL til å avgjøre om det finnes en sykel i sammenhengende komponent til start -noden.

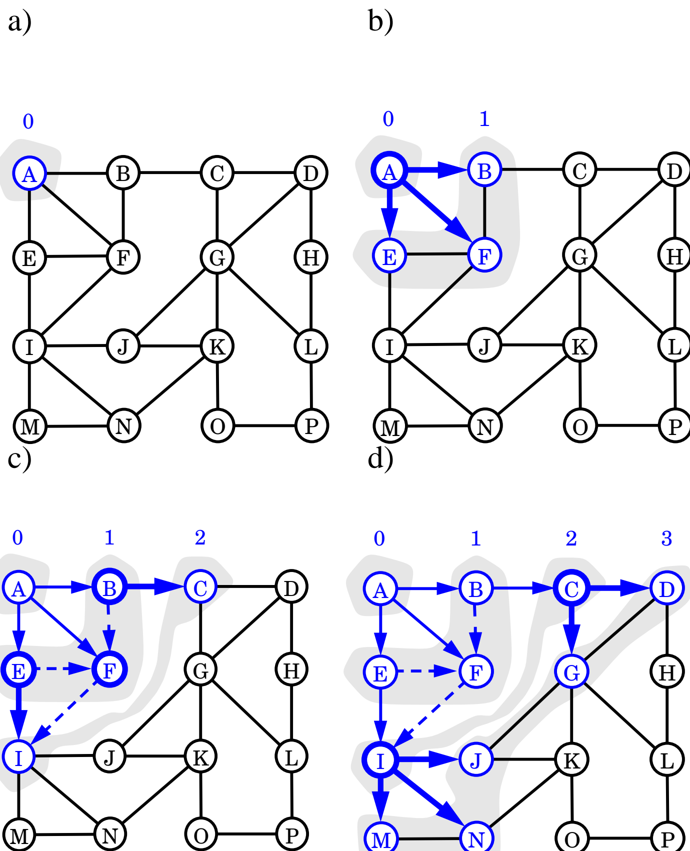
```
public class FindCycleDFS extends DFSMAL {
  protected Sequence path;
  protected boolean done;
  protected Vertex cycleStart;
  public Object execute(InspectableGraph g, Vertex start,
                        Object info) {
    super.execute(g, start, info);
    path = new NodeSequence();
    done = false;
    dfsVisit(start);
    //copy the vertices up to cycleStart from the path to
    //the cycle sequence.
    Sequence theCycle = new NodeSequence();
    Enumeration pathVerts = path.elements();
```

```
      while (pathVerts.hasMoreElements()) {
         Vertex v = (Vertex)pathVerts.nextElement();
         theCycle.insertFirst(v);
         if ( v == cycleStart) {
            break;
         }
      }
      return theCycle.elements();
   }
   protected void startVisit(Vertex v) {path.insertFirst(v);}
   protected void finishVisit(Vertex v) {
      if (done) {path.remove(path.first());}
   }
   //When a back edge is found, the graph has a cycle
   protected void traverseBack(Edge e, Vertex from) {
      Enumeration pathVerts = path.elements();
      cycleStart = graph.opposite(from, e);
      done = true;
   }
   protected boolean isDone() {return done;}
}
```

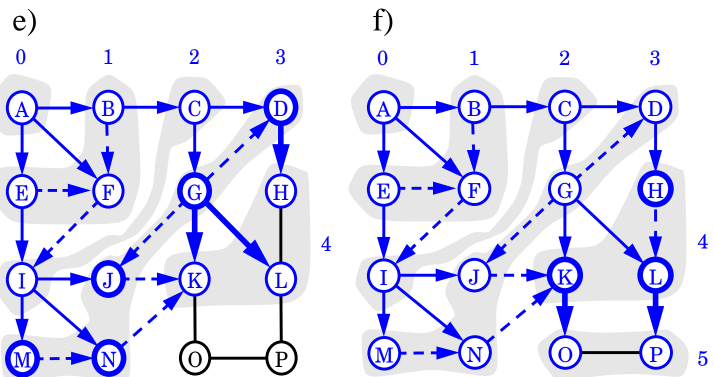

Bredde-Først Søk

- Et Bredde-Først Søk (BFS) traverserer også den sammhengende komponent, og definerer også et utspennende tre hvor nodene legges i nivåer:
  - Startnoden *s* har nivå 0,
  - I første runde besøker vi alle noder som kan nåes på en sti av lengde 1 fra startnoden (dvs de er naboer til startnoden), og alle disse nodene ligger på nivå 1
  - I andre runde besøker vi alle noder som kan nåes på en sti av lengde 2 fra startnoden (dvs disse er naboer av noder på nivå 1), og alle disse nodene ligger på nivå 2.
  - Osv inntill alle noder blir tildelt et nivå som vil tilsvare antall kanter på korteste sti fra startnode til noden.

BFS - eksempel



BFS



BFS Pseudokode

```
Algorithm BFS(s):
  Input: En node s i en graf
  Output: Merking av kanter som “discovery”-kanter
    eller “kryss”-kanter
  container L0 = {s}
  i ← 0
  while Li ikke-tom do
    ny tom container Li+1
    for hver node u i Li do
      for hver kant e=(u,v) incident med u do
        if node v ikke-besøkt then
          merk e som discovery-kant
          settinn w i Li+1
        else
          merk e som kryss-kant
  i ← i + 1
```


BFS-Egenskaper

- Proposisjon: La G være en urettet graf hvor vi kjører BFS med startnode s. Da får vi at
  - BFS besøker alle noder i den sammenhengende komponent til s.
  - discovery-kantene utgjør et utspennende tre T, som vi kaller BFS -treet.
  - For hver node v på nivå i i treet, så er det i kanter på den korteste sti fra s til denne noden.
  - If (u, v) ikke er i BFS treet, så er forskjellen på nivåene til u og v max 1.
- Proposisjon: La G være graf med n noder og m kanter. En BFS traversering av G tar tid $O(n + m)$. Spesilaiseringer av BFS gir $O(n + m)$ tidsalgoritmer for:
  - Teste om G sammenhengende.
  - Finne utspennende tre til G
  - Finne alle sammenhengende komponenter til G
  - Finne, for hver node v i G, det minste antall kanter på en sti mellom s og v.