5SENG003W - Algorithms, Week 8

Dr. Klaus Draeger

RECAP

Last week...

- We talked about some more specialised data structures
  - Stacks
    - Last In, First Out
  - Queues
    - First In, First Out
  - Priority queues
    - Contents partially sorted by priority
    - Highest priority item in front
  - Heaps
    - Used in priority queues
    - Heap Sort

Overview of today's lecture

- Graphs
  - Definition
  - Representations
  - Properties
- Graph traversals
  - Breadth-first
  - Depth-first

Introduction to Graphs

Graphs are a more general non-linear data structure than trees.
A graph $G = (V, E)$ is given by:

- A set V of **vertices** (or nodes)
- A set E of **edges**

where edges represent connections between vertices, either:

- **directed** edges **from** $v \in V$ **to** $w \in V$
- **undirected** edges **between** $v \in V$ and $w \in V$

Accordingly G is a **directed** or **undirected** graph.

- Each vertex is **incident** with the edges connected to it
- Two vertices are **adjacent** if they share an edge.

The **degree** of a vertex v is the number of incident edges. In a directed graph this is split into

- the **indegree**: the number of edges **into** v,
- the **outdegree**: the number of edges **out of** v

Example: Directed and Undirected Graphs

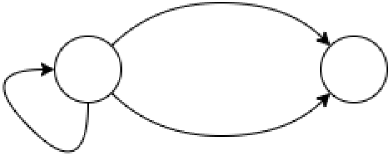
These graphs are defined by:

- $V_1 = \{a, b, c, d\}$,
  $E_1 = \{(a, b), (a, c), (b, d), (c, b), (d, c)\}$
  - Directed edges written as **ordered** pairs like (a, b)
  - Note (a, b) is **different** from (b, a)
- $V_2 = \{e, f, g, h\}$,
  $E_2 = \{\{e, f\}, \{e, h\}, \{f, h\}, \{g, h\}\}$
  - Undirected edges written as **unordered** pairs like $\{e, f\}$
  - Note $\{e, f\}$ is **the same** as $\{f, e\}$

Loops and Parallel Edges

Some graphs are more general than what we have defined

- There can be **loops** from a vertex to itself
  - This can still be represented as a pair (v, v) or $\{v, v\}$
- There can be **parallel edges**, i.e. multiple edges between the same vertices
  - For this we would use separate sets $V = \{v_1, v_2, \ldots, v_k\}$ and $E = \{e_1, e_2, \ldots, e_n\}$
  - Along with a function mapping each edge to a pair of vertices



Additional Data

The vertices and/or edges can have additional data attached to them

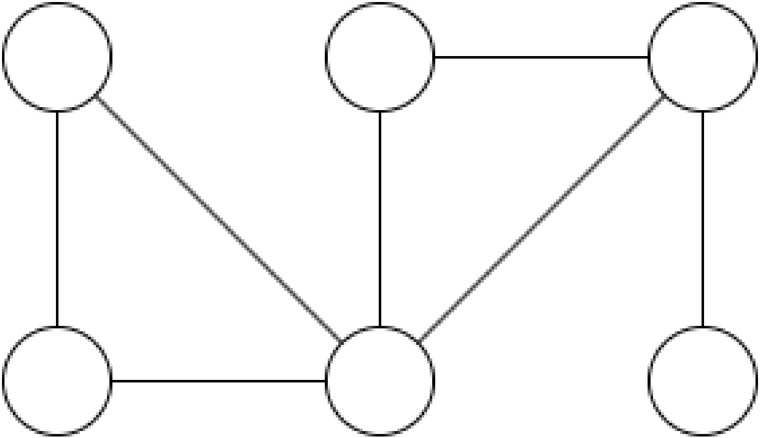
- Both as part of the input and the solution
- Either **discrete** data
  - Colours in **colouring problems**
  - Underground lines on the tube map
  - Transition labels in **finite state machines**
- Or infinite domains like the integers
  - Weights (distances, costs), giving rise to **weighted graphs**
  - Capacities in **network flow** problems
- We will have a quick look at some example problems.

Example: Colouring problems

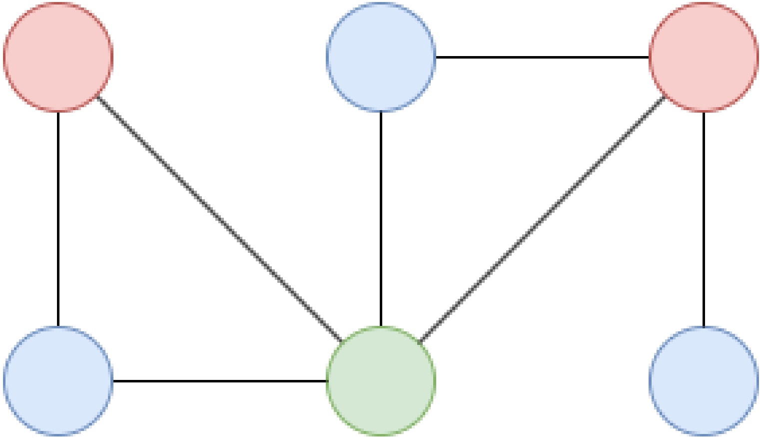
Colouring problems are about assigning colours to the vertices or edges of an undirected graph

- Edge colouring:
  - Assign a colour from a given set to each **edge**
  - At each vertex, all incident edges must have different colours
- Vertex colouring:
  - Assign a colour from a given set to each **vertex**
  - Adjacent verteices must have different colours
  - Originates in map drawing
    - Assign one vertex per country
    - Add adges between neighbouring countries
    - These should have different colours to be distinguishable
    - Usually want to use as few colours as possible

Graph Colouring Example



Graph Colouring Example



Example: Maximum Flow

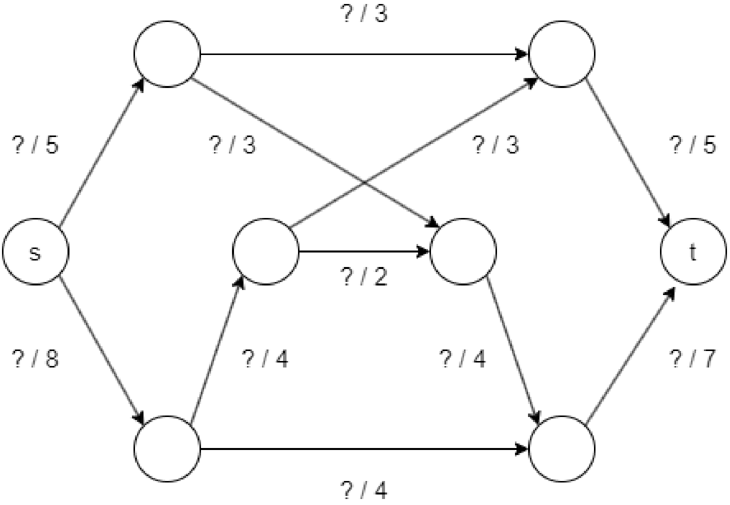
In a **Maximum Flow** problem we want to get a resource from a **source** to a **target** (or **sink**) within a network.

- The network is a directed graph $G = (V, E)$
- Every edge $e \in E$ has a **capacity** $c(e)$
- The source and target are vertices in V

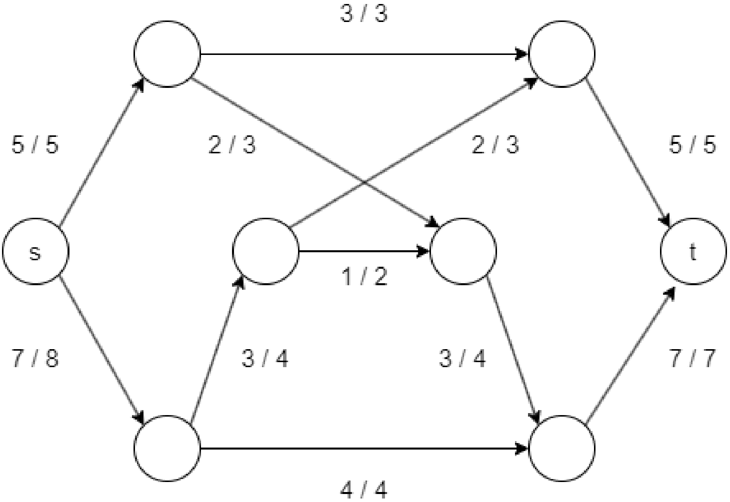
We want to assign to each edge e another number, the **flow** $f(e)$ such that

- $f(e)$ is between 0 and $c(e)$
- At every vertex, the total flow in is the same as the total flow out
  - except at the source (only flow out) and target (only flow in)
- The total flow into the target is as high as possible

Maximum Flow Example



Maximum Flow Example



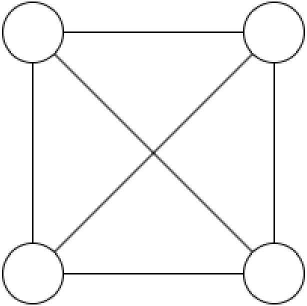
Some Graph Properties

Let $G = (V, E)$ be a graph.

- We will write
  - $v \to w$ if (v, w) or $\{v, w\}$ is in E
  - $v \leftrightarrow w$ if (v, w), (w, v) or $\{v, w\}$ is in E
- A sequence of vertices $v_1, v_2, \ldots, v_n \in V$ is
  - a **walk** if $v_1 \to v_2, \ldots, v_{n-1} \to v_n$
  - a **lax walk** if $v_1 \leftrightarrow v_2, \ldots, v_{n-1} \leftrightarrow v_n$
    (i.e. it ignores edge directions)
- It is a **path** if it has no repeated vertices.
- It is a **cycle** if it has more than one vertex and its first and last vertex are the same.
- An **Euler Walk** is a walk using every edge exactly once.
- A **Hamiltonian Path** is a path visiting every vertex exactly once.

Euler Walk Example

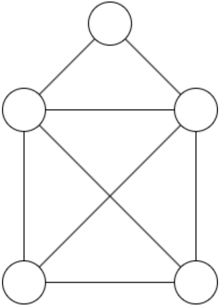
This graph does **not** have an Euler Walk:



- Each vertex except the first and last would be exited as often as it would be entered
- So each vertex except the first and last would have to have **even** degree
- But here all vertices have degree 3

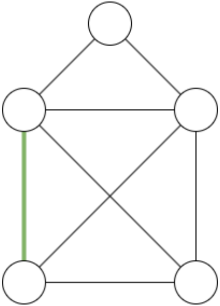
Euler Walk Example

This graph **does** have an Euler Walk:



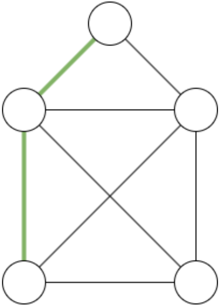
Euler Walk Example

This graph **does** have an Euler Walk:



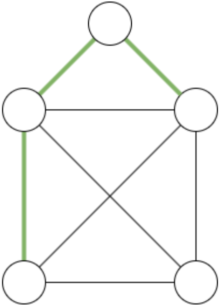
Euler Walk Example

This graph **does** have an Euler Walk:



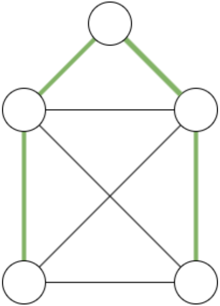
Euler Walk Example

This graph **does** have an Euler Walk:



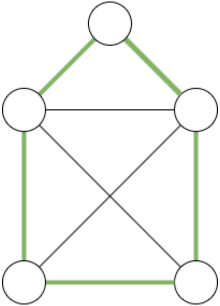
Euler Walk Example

This graph **does** have an Euler Walk:



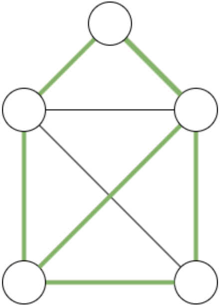
Euler Walk Example

This graph **does** have an Euler Walk:



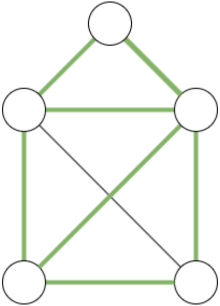
Euler Walk Example

This graph **does** have an Euler Walk:



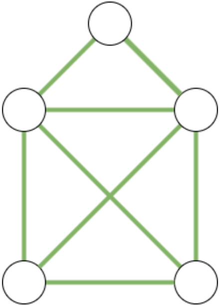
Euler Walk Example

This graph **does** have an Euler Walk:



Euler Walk Example

This graph **does** have an Euler Walk:



Some Graph Properties

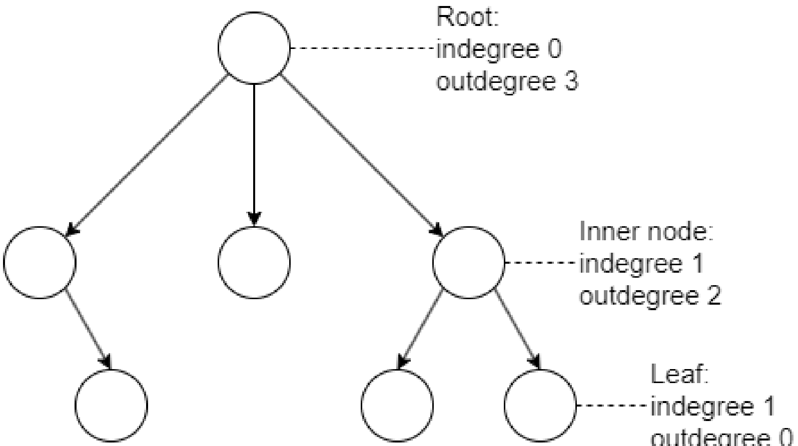
Based on these we can define properties of a graph:

- G is **acyclic** if it has no cycles
- G is **connected** if there is a lax walk between any two vertices
- Equivalently: G is connected if we cannot split V into nonempty sets $V = V_1 \cup V_2$ such that there are no edges between V_1 and V_2
- A **directed** graph is **strongly connected** if there is a path between any two vertices.

We can also define trees in several ways, e.g.:

- An **undirected** graph is a tree if it is connected and acyclic
- A **directed** graph is a (rooted) tree if
  - it is connected
  - no vertex has an indegree > 1
    (then the root has indegree 0 and leaves have outdegree 0)

Example


Root:
indegree 0
outdegree 3
Inner node:
indegree 1
outdegree 2
Leaf:
indegree 1
outdegree 0

Representing graphs

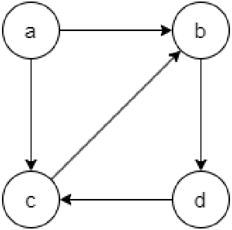
There are two main ways of representing a graph:

- The **adjacency matrix**
  - This is an $n * n$ matrix where n is the size of V
  - Entries are $1, 0$ (or general numbers for **weighted** graphs)
  - It will be **symmetric** for **undirected** graphs
- The **adjacency lists**
  - **Vertex** class
  - Each vertex contains a list
    - Can be **linked** or **array-based**
    - Contains either **vertices** or **edges** (using a separate class) if they need to contain additional data (weights etc)

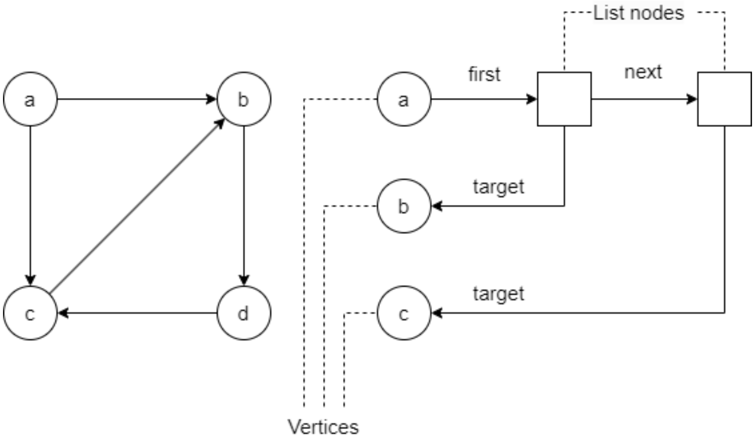
Adjacency Matrix Example

$$\begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

is the adjacency matrix of this graph:



Adjacency List Example



Comparison

Adjacency matrices are

- easy to implement
- memory efficient for **dense** graphs
- convenient for some computations
  - e.g. to compute the **numbers of paths** of length k, compute A^k

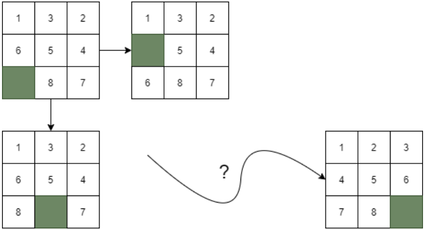
Adjacency lists are

- memory efficient for **sparse** graphs
- suitable in case the number of vertices is initially **unknown** – why would this be?

Example: State-Transition Graphs

One major application for graphs is as a representation of systems:

- Vertices are **states**
- Edges are **transitions**



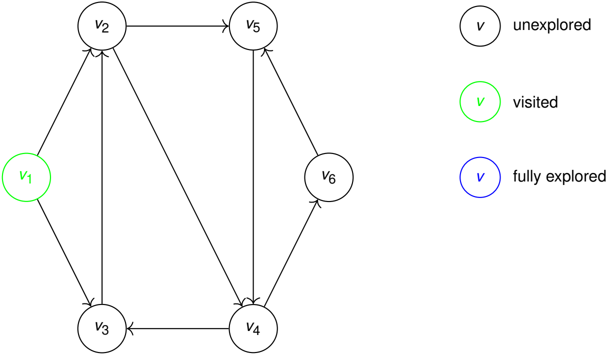
- These can be very large (3602880 states even for this simple 3*3 puzzle)
- We ideally want to represent only those which are needed
- This leads to "on the fly" exploration

Searching in graphs

- A common operation on graphs is **search**
  - Searching for some particular **vertex**
  - Searching for any vertex satisfying a given **condition**
  - Searching for a **path** between two vertices
  - Searching for a **shortest** path between two vertices
- Search in graphs is more complex than in trees or lists
  - Graphs can have **cycles**
  - Naive search could get stuck in a loop
  - Need to keep track of visited vertices to prevent this
- Two main strategies: **depth-first** and **breadth-first** search

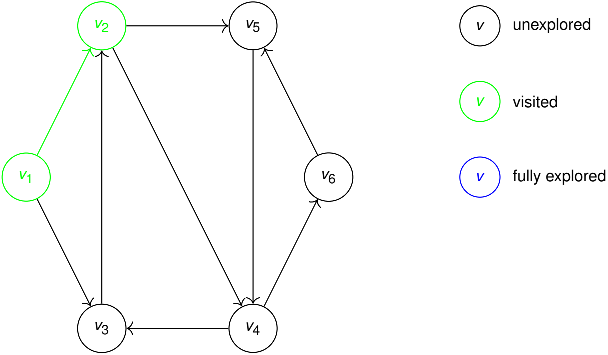
Depth-first search

- In depth-first search, we recursively follow outgoing edges



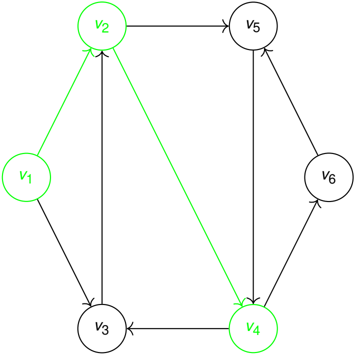
Depth-first search

- In depth-first search, we recursively follow outgoing edges



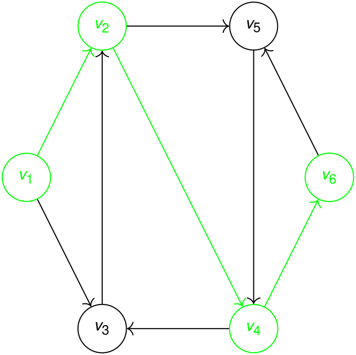
Depth-first search

- In depth-first search, we recursively follow outgoing edges



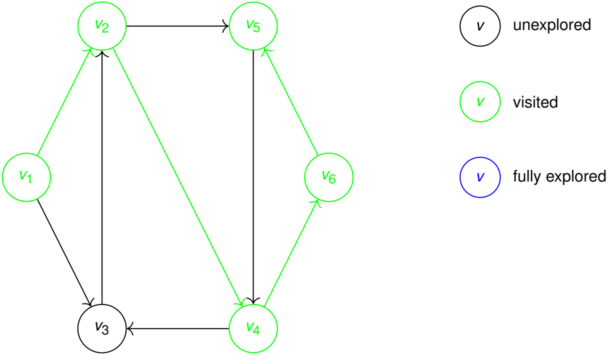
Depth-first search

- In depth-first search, we recursively follow outgoing edges



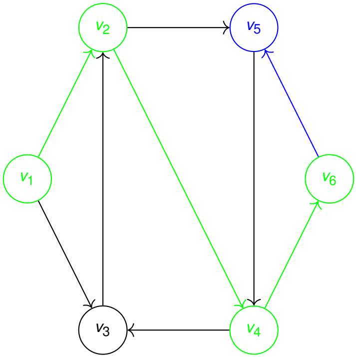
Depth-first search

- In depth-first search, we recursively follow outgoing edges



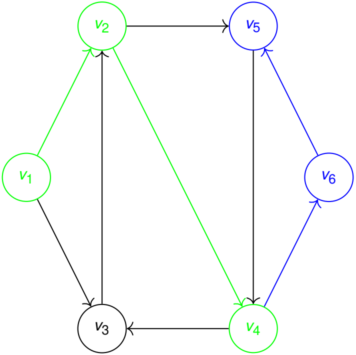
Depth-first search

- In depth-first search, we recursively follow outgoing edges



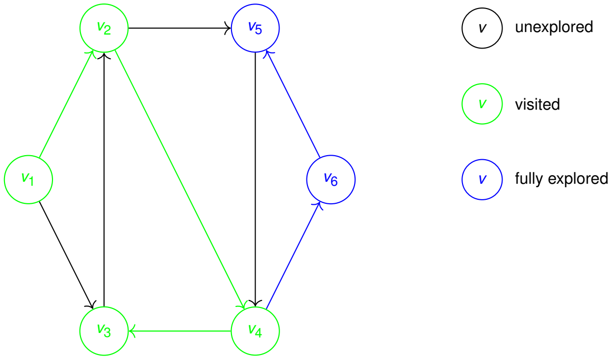
Depth-first search

- In depth-first search, we recursively follow outgoing edges



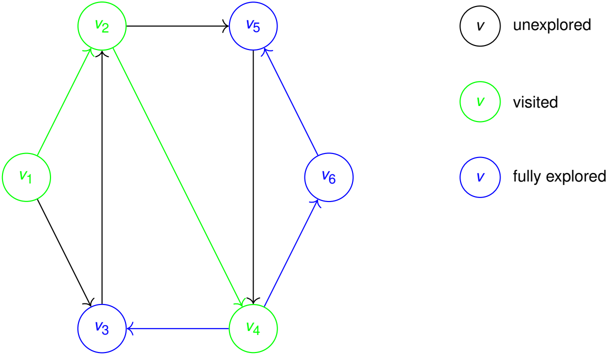
Depth-first search

- In depth-first search, we recursively follow outgoing edges



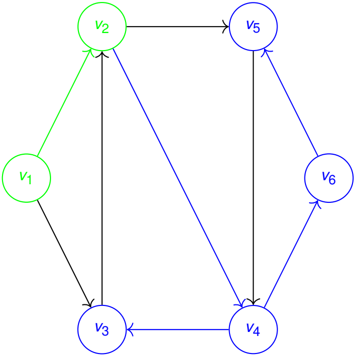
Depth-first search

- In depth-first search, we recursively follow outgoing edges



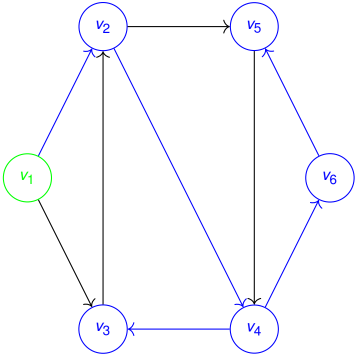
Depth-first search

- In depth-first search, we recursively follow outgoing edges



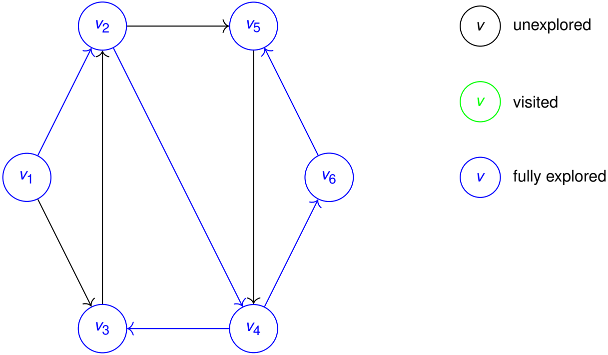
Depth-first search

- In depth-first search, we recursively follow outgoing edges



Depth-first search

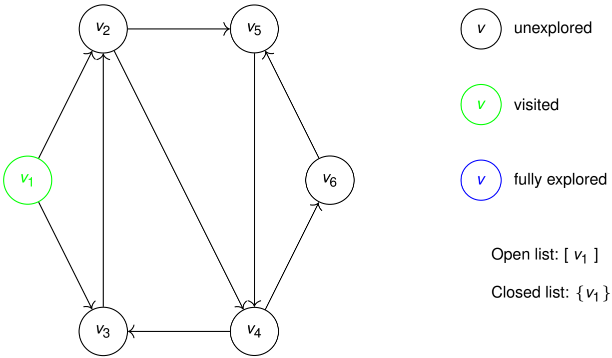
- In depth-first search, we recursively follow outgoing edges



Recap: Breadth-first search

- In breadth-first search, we explore the graph in "layers"
- Not recursive
- We maintain two data structures:
  - The set of all visited vertices
    (often called the **"closed list"**)
  - A queue of vertices that we have visited but not yet fully explored
    (often called the **"open list"**)
- In each iteration, we
  - Go through the edges out of the open list's front element
  - Enqueue those edge targets that are not in the closed list
    (and add them to the closed list)
  - Dequeue the front element (it is now fully explored)

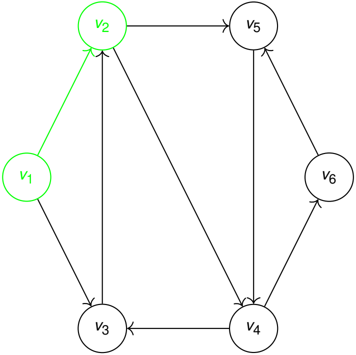
Recap: Breadth-first search



Open list: [v_1]

Closed list: $\{v_1\}$

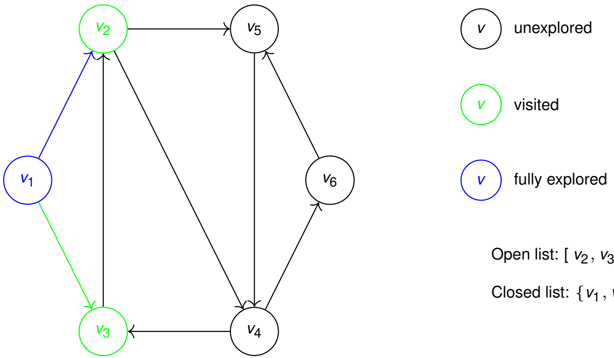
Recap: Breadth-first search



Open list: [v_1 , v_2]

Closed list: $\{v_1 , v_2\}$

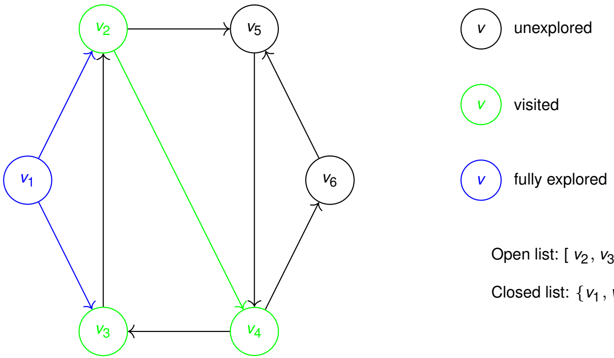
Recap: Breadth-first search



Open list: [v_2, v_3]

Closed list: $\{v_1, v_2, v_3\}$

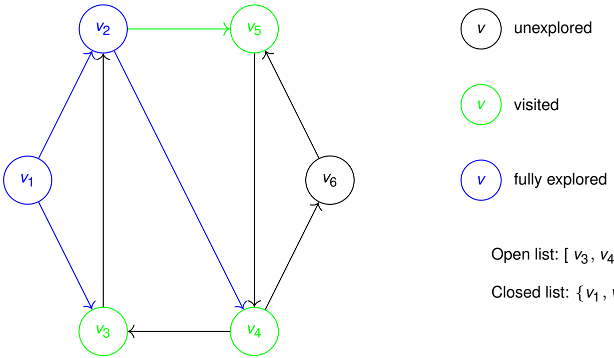
Recap: Breadth-first search



Open list: [v_2, v_3, v_4]

Closed list: $\{v_1, v_2, v_3, v_4\}$

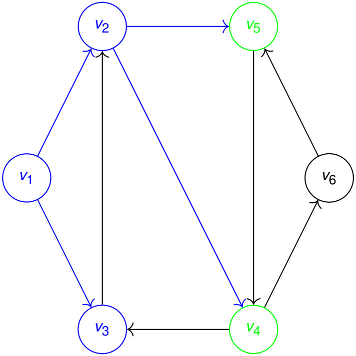
Recap: Breadth-first search



Open list: [v_3, v_4, v_5]

Closed list: $\{v_1, v_2, v_3, v_4, v_5\}$

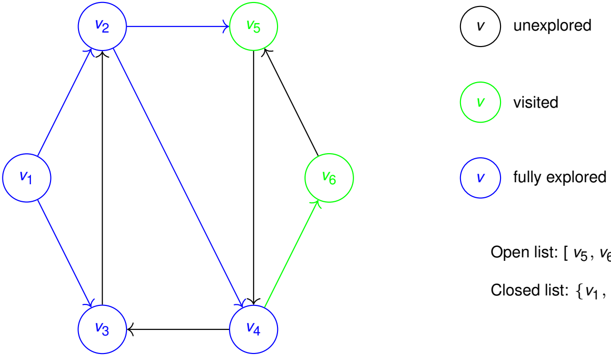
Recap: Breadth-first search



Open list: [v_4, v_5]

Closed list: $\{v_1, v_2, v_3, v_4, v_5\}$

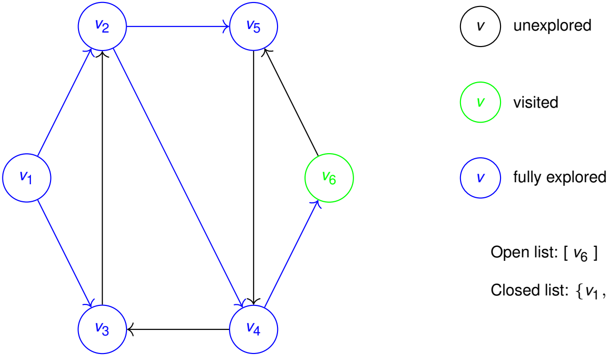
Recap: Breadth-first search



Open list: [v_5, v_6]

Closed list: $\{v_1, v_2, v_3, v_4, v_5, v_6\}$

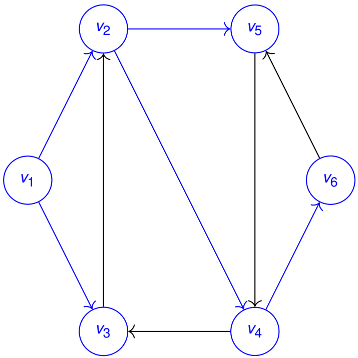
Recap: Breadth-first search



Open list: [v_6]

Closed list: $\{v_1, v_2, v_3, v_4, v_5, v_6\}$

Recap: Breadth-first search

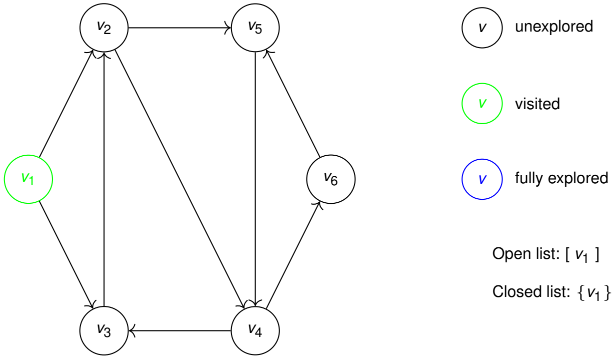


Open list: [ ]

Closed list: $\{v_1, v_2, v_3, v_4, v_5, v_6\}$

Depth-first search: iterative version

- We can also perform depth-first search in the iterative style that we saw in breadth-first search
- The key change is organising the open list as a **stack** rather than a queue

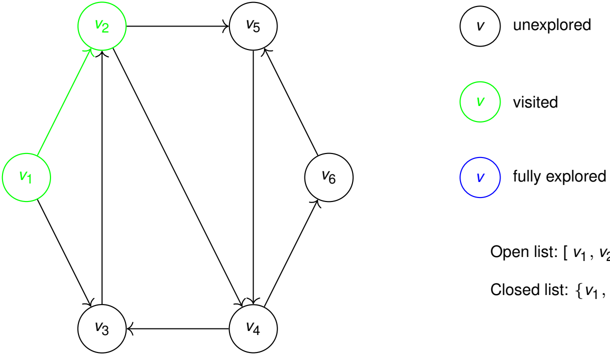


Open list: [v_1]

Closed list: $\{v_1\}$

Depth-first search: iterative version

- We can also perform depth-first search in the iterative style that we saw in breadth-first search
- The key change is organising the open list as a **stack** rather than a queue

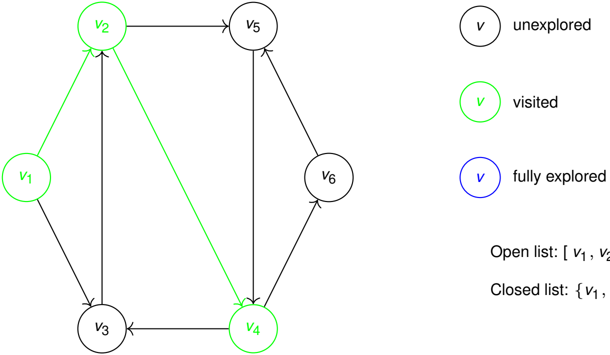


Open list: [v_1 , v_2]

Closed list: $\{v_1, v_2\}$

Depth-first search: iterative version

- We can also perform depth-first search in the iterative style that we saw in breadth-first search
- The key change is organising the open list as a **stack** rather than a queue

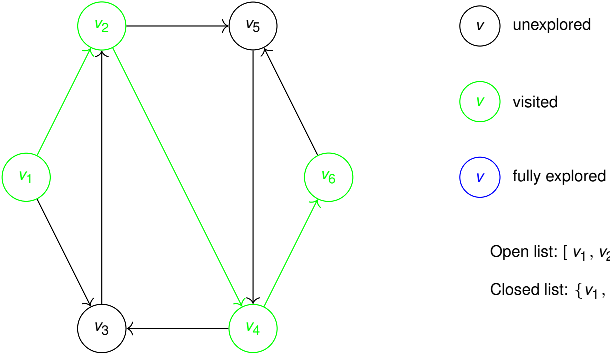


Open list: [v_1 , v_2 , v_4]

Closed list: $\{v_1, v_2, v_4\}$

Depth-first search: iterative version

- We can also perform depth-first search in the iterative style that we saw in breadth-first search
- The key change is organising the open list as a **stack** rather than a queue

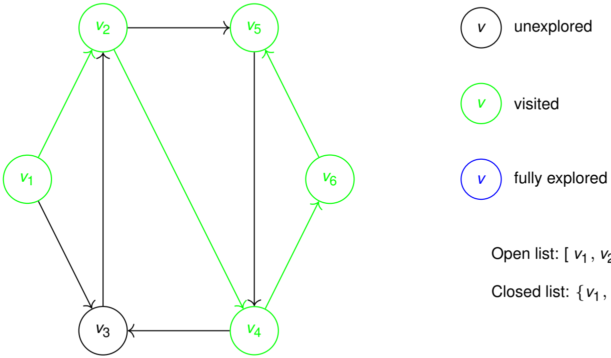


Open list: [v_1, v_2, v_4, v_6]

Closed list: $\{v_1, v_2, v_4, v_6\}$

Depth-first search: iterative version

- We can also perform depth-first search in the iterative style that we saw in breadth-first search
- The key change is organising the open list as a **stack** rather than a queue

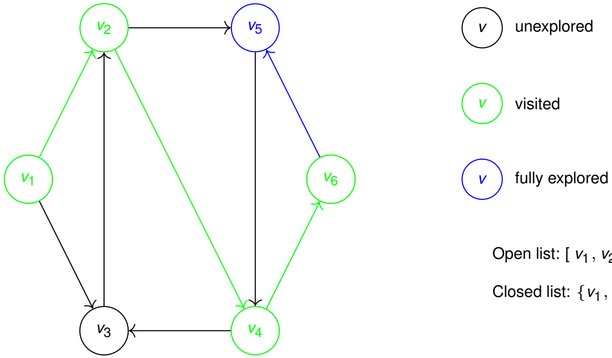


Open list: [v_1, v_2, v_4, v_6, v_5]

Closed list: $\{v_1, v_2, v_4, v_5, v_6\}$

Depth-first search: iterative version

- We can also perform depth-first search in the iterative style that we saw in breadth-first search
- The key change is organising the open list as a **stack** rather than a queue

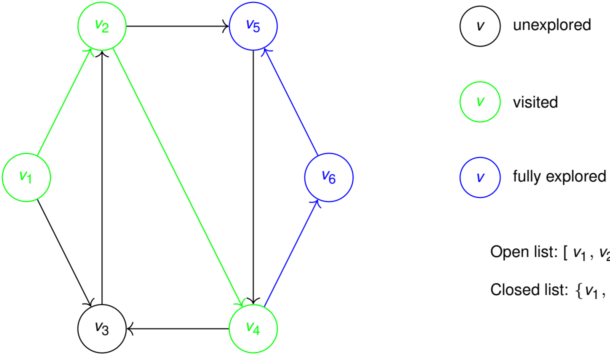


Open list: [v_1, v_2, v_4, v_6]

Closed list: $\{v_1, v_2, v_4, v_5, v_6\}$

Depth-first search: iterative version

- We can also perform depth-first search in the iterative style that we saw in breadth-first search
- The key change is organising the open list as a **stack** rather than a queue

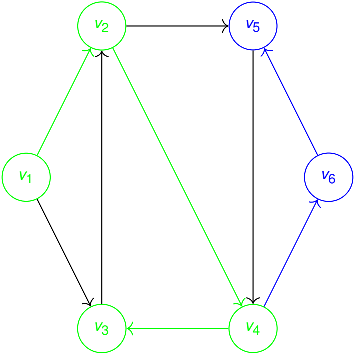


Open list: [v_1 , v_2 , v_4]

Closed list: $\{v_1, v_2, v_4, v_5, v_6\}$

Depth-first search: iterative version

- We can also perform depth-first search in the iterative style that we saw in breadth-first search
- The key change is organising the open list as a **stack** rather than a queue

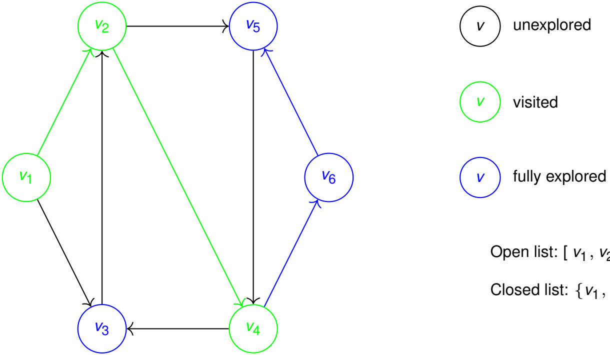


Open list: [v_1, v_2, v_4, v_3]

Closed list: $\{v_1, v_2, v_3, v_4, v_5, v_6\}$

Depth-first search: iterative version

- We can also perform depth-first search in the iterative style that we saw in breadth-first search
- The key change is organising the open list as a **stack** rather than a queue

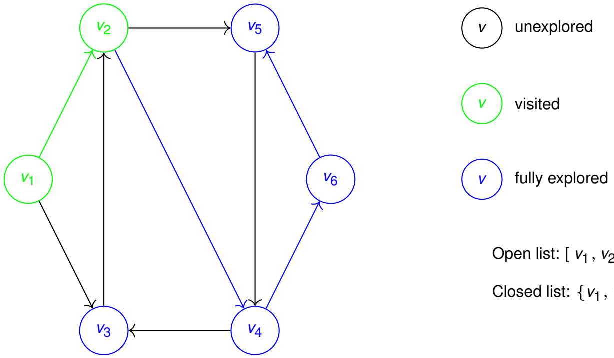


Open list: [v_1, v_2, v_4]

Closed list: $\{v_1, v_2, v_3, v_4, v_5, v_6\}$

Depth-first search: iterative version

- We can also perform depth-first search in the iterative style that we saw in breadth-first search
- The key change is organising the open list as a **stack** rather than a queue

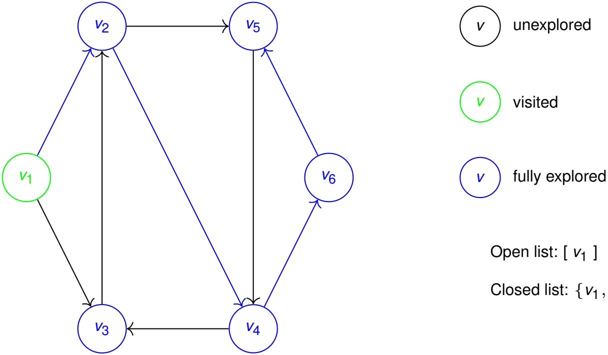


Open list: [v_1, v_2]

Closed list: $\{v_1, v_2, v_3, v_4, v_5, v_6\}$

Depth-first search: iterative version

- We can also perform depth-first search in the iterative style that we saw in breadth-first search
- The key change is organising the open list as a **stack** rather than a queue

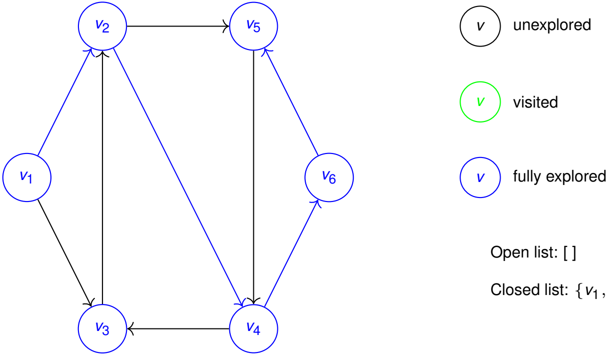


Open list: [v_1]

Closed list: $\{v_1, v_2, v_3, v_4, v_5, v_6\}$

Depth-first search: iterative version

- We can also perform depth-first search in the iterative style that we saw in breadth-first search
- The key change is organising the open list as a **stack** rather than a queue



Open list: [ ]

Closed list: $\{v_1, v_2, v_3, v_4, v_5, v_6\}$

Depth-first search: iterative version

- The iterative version of DFS looks like this (assuming for simplicity that Vertex contains **`List<Vertex> adjacencyList;`**)

```
// Try to find an edge v->w with w not yet visited
public Vertex findSuccessor(Vertex v, Set<Vertex> c){
   for(Vertex w : v.adjacencyList)
      if(! c.contains(w))  // not yet visited
         return w;
   return null;  // no suitable edge
}

public void DFS(Vertex source){
   // Set up open and closed list
   Stack<Vertex> o = new Stack<Vertex>();
   HashSet<Vertex> c = new HashSet<Vertex>();
   o.push(source);
   c.add(source);
   while(!o.empty()){
      Vertex v = o.peek(); // vertex on top of the stack
      Vertex w = findSuccessor(v, c);
      if(w != null){       // push w onto o and add it to c
         o.push(w);
         c.add(w);
      }
      else    // v is a dead end; pop it off o
         o.pop();
   }
}
```