5SENG003W - Algorithms, Week 9

Dr. Klaus Draeger

RECAP: Graphs

- Graphs consist of **vertices** connected by **edges**
- Edges can be **directed** or **undirected**
- Representations:
  - Set of vertex pairs
    - Ordered pairs for directed graphs: $\{(v_1, v_2), (v_1, v_3), (v_2, v_5), \ldots\}$
    - Unordered pairs for undirected graphs: $\{\{v_1, v_7\}, \{v_2, v_5\}, \{v_2, v_3\}, \ldots\}$
  - Adjacency matrix
    - **`a[i][j]`** is 1 if there is an edge (v_i, v_j) and 0 otherwise
  - Adjacency lists
    - Each vertex contains a list of outgoing edges
    - Vertices can be implicit: can represent the graph by an array of lists

Recap: Graph properties

- **Adjacency** between vertices, **incidence** between vertices and edges
- (In-, out-) degree
- Walks, paths, cycles
- Loops and parallel edges
- Connectedness, acyclicity
- Colours, weights, ...

More about graph representations

- We have previously seen that graphs can be represented using
  - Adjacency **matrices**
  - Adjacency **lists**
- Let us talk about this in more detail. The main questions to ask yourself are
  - How do we represent **vertices**?
  - How do we represent **edges**?
  - how do we store **additional data** (labels, weights, ...)?

Representing vertices

- We can represent vertices using
  - Explicit **`Vertex`** objects
    - Containing the adjacency list (makes no sense with an adjacency matrix)
    - Containing all relevant additional data
    - Suitable if number of vertices is initially unknown
  - IDs
    - To index the adjacency matrix row
      or the adjacency list (in the array of lists)
    - Could be more than a single number (e.g. co-ordinates on a chessboard)

Representing edges

- We can represent edges using
  - Explicit **`Edge`** objects
    - Containing the target verted
    - Containing all relevant additional data
    - Suitable if there are parallel edges or complex additional data
  - Just the target vertex
    - ID or pointer to object

Recap: Searching in graphs

- A common operation on graphs is **exploration**
  - Searching for some particular **vertex**
  - Searching for a **path** between two vertices
  - Searching for a **shortest** path between two vertices
  - Or just trying to **fully explore** the graph
- Graph exploration 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
- Previously introduced: **depth-first** and **breadth-first** search

Recap: Depth-first search

- In depth-first search, we recursively follow outgoing edges
- To explore a vertex, we
  - Pick a previously unvisited neighbour
  - Recursively explore that neighbour
  - Repeat if (after returning from the recursive call) there are still unvisited neighbours
- Note the difference between **visited** and **explored**
  - A node is **visited** as soon as we encounter it
  - A node is **explored** once we have visited all its neighbours (possibly from a different node)
  - This difference will be represented by the **closed** and **open** lists

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
    (the **closed list**)
  - A queue of visited but not yet fully explored vertices
    (the **open list**)
- In each iteration, we
  - Check the neighbours of the open list's front element
  - Enqueue those which were not yet visited
    (and add them to the closed list)
  - Dequeue the front element (it is now fully explored)

Recap: Iterative depth-first search

- We can perform depth-first search in the same iterative style as breadth-first search
- The key change is organising the open list as a **stack** rather than a queue
- In each iteration, we
  - Check if the top element has unvisited neighbours
  - If yes, push one of them onto the open list (and add it to the closed list)
  - If not, pop the top element off (it is now fully explored)
- Alternatively, in each iteration, we
  - Pop the top element of the stack
  - Push **all** its unvisited neighbours onto the open list (and add them to the closed list)
- The first version means re-examining adjacency lists but will directly produce a path to the target (if any)

Depth-first search: iterative version

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

```
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.pop();
        for(Vertex w : v.adjacencyList)
            if(! c.contains(w)){  // not yet visited
                o.push(w);
                c.add(w);
            }
    }
}
```

Aside: eliminating recursion

- Iterative DFS is an example of a more general idea: We can get rid of recursion by using a stack (to store pending branches)
- Example: pre-order traversal in a tree looks like this:

```
public class TreeNode{
    public int data;
    public TreeNode leftChild, rightChild;
}

public void preOrder(TreeNode n){
    if(n != null){
        System.out.println(n.data);
        preOrder(n.leftChild);
        preOrder(n.rightChild);
    }
}
```

Aside: eliminating recursion

- Example: an **iterative** version of pre-order traversal (using java.util.Stack)

```
public class TreeNode{
    public int data;
    public TreeNode leftChild, rightChild;
}

public void preOrder(TreeNode t){
    Stack<TreeNode> s = new Stack<TreeNode>();
    s.push(t);
    while(!s.empty()){
        // pop top branch off the stack
        t = s.pop();
        if(t != null){  // output data and push children on the stack
            System.out.println(t.data);
            s.push(t.rightChild);
            s.push(t.leftChild);
        }
    }
}
```

- This code could still be improved (some vertices get pushed and then immediately popped)
- Think about how to do this with in-order or post-order

Exploration for pathfinding

- The traversal algorithms explore the entire graph
- When searching for a path to a target this is not needed. Instead we can
  - Abort the search as soon as the target is found
  - Keep track of the path taken, e.g. using a **predecessor map**
  - This map represents a **spanning tree** with predecessors being parents
- Either algorithm will find a path if one exists
- BFS also guarantees that the path found has **minimal length**

Why breadth-first search works

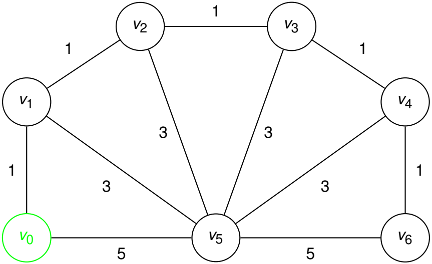
- BFS advances in "layers":
  - For a vertex v, let $f(v)$ be the distance from the start to v.
  - BFS first visits first all v with $f(v) = 0$, then $f(v) = 1$ etc
- Why is this the case?
  - At first, the open list only contains the start vertex, which is the only vertex s with $f(s) = 0$
  - All vertices with $f(v) = k + 1$ are reached in one step from vertices with $f(v) = k$
  - So they are all enqueued while the vertices with $f(v) = k$ are getting visited

Breadth-first search breaks for weighted graphs

- In a weighted graph, vertices are no longer found ordered by distance
- For example, vertices with $f(v) = 5$ may have edges to vertices with $f(v) = k$ for **any** $k > 5$.
- Idea: could use a **sorted** data structure like a binary search tree for the open list to fix this
  - Then insertion and extraction in the open list both are in $O(\log n)$
- But we only ever need the minimal-distance vertex
  - So we can use a **priority queue** based on a **min-heap**.
- This gives us **Dijkstra's Algorithm.**
  - Finds shortest paths from the start to all other targets
  - Easy to turn into a single-target version

Dijkstra's algorithm: example

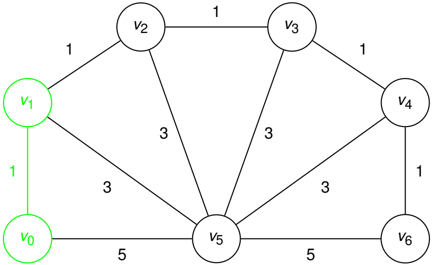
- Consider this example:



- Open list:
  v_0
- Closed list:
  $\{v_0 \mapsto 0\}$
- Step 5: Vertex v_2 inserted in front of v_5
- Done; found shortest distances to all vertices

Dijkstra's algorithm: example

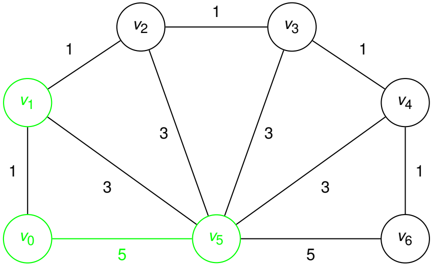
- Consider this example:



- Open list:
  v_0 v_1
- Closed list:
  $\{v_0 \mapsto 0, v_1 \mapsto 1\}$
- Step 5: Vertex v_2 inserted in front of v_5
- Done; found shortest distances to all vertices

Dijkstra's algorithm: example

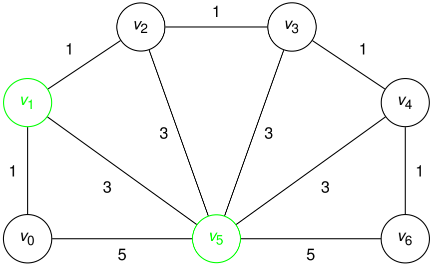
- Consider this example:



- Open list:
  v_0 v_1 v_5
- Closed list:
  $\{v_0 \mapsto 0, v_1 \mapsto 1, v_5 \mapsto 5\}$
- Step 5: Vertex v_2 inserted in front of v_5
- Done; found shortest distances to all vertices

Dijkstra's algorithm: example

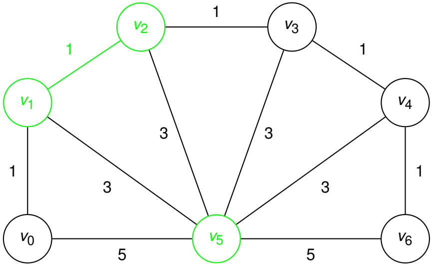
- Consider this example:



- Open list:
  v_1 v_5
- Closed list:
  $\{v_0 \mapsto 0, v_1 \mapsto 1, v_5 \mapsto 5\}$
- Step 5: Vertex v_2 inserted in front of v_5
- Done; found shortest distances to all vertices

Dijkstra's algorithm: example

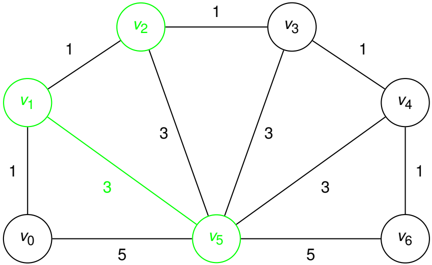
- Consider this example:



- Open list:
  v_1 v_2 v_5
- Closed list:
  $\{v_0 \mapsto 0, v_1 \mapsto 1, v_2 \mapsto 2, v_5 \mapsto 5\}$
- Step 5: Vertex v_2 inserted in front of v_5
- Done; found shortest distances to all vertices

Dijkstra's algorithm: example

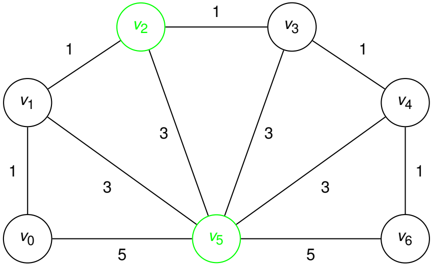
- Consider this example:



- Open list:
  v_1 v_2 v_5
- Closed list:
  $\{v_0 \mapsto 0, v_1 \mapsto 1, v_2 \mapsto 2, v_5 \mapsto 4\}$
- Step 5: Vertex v_2 inserted in front of v_5
- Done; found shortest distances to all vertices

Dijkstra's algorithm: example

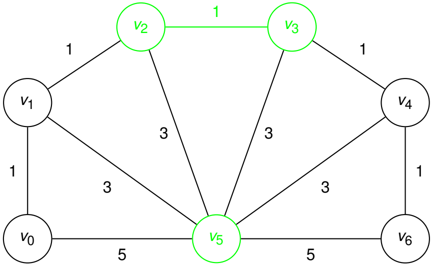
- Consider this example:



- Open list:
  v_2 v_5
- Closed list:
  $\{v_0 \mapsto 0, v_1 \mapsto 1, v_2 \mapsto 2, v_5 \mapsto 4\}$
- Step 5: Vertex v_2 inserted in front of v_5
- Done; found shortest distances to all vertices

Dijkstra's algorithm: example

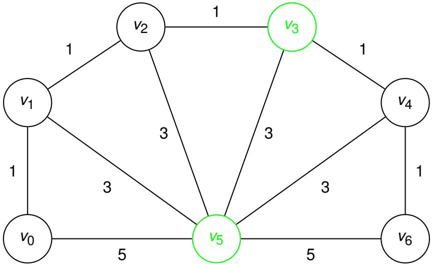
- Consider this example:



- Open list:
  $v_2\ v_3\ v_5$
- Closed list:
  $\{v_0 \mapsto 0, v_1 \mapsto 1, v_2 \mapsto 2, v_3 \mapsto 3, v_5 \mapsto 4\}$
- Step 5: Vertex v_2 inserted in front of v_5
- Done; found shortest distances to all vertices

Dijkstra's algorithm: example

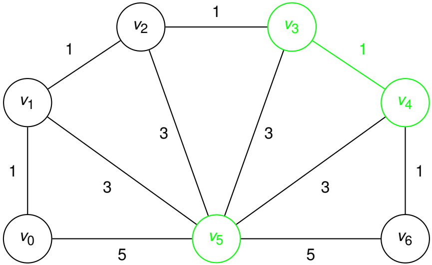
- Consider this example:



- Open list:
  v_3 v_5
- Closed list:
  $\{v_0 \mapsto 0, v_1 \mapsto 1, v_2 \mapsto 2, v_3 \mapsto 3, v_5 \mapsto 4\}$
- Step 5: Vertex v_2 inserted in front of v_5
- Done; found shortest distances to all vertices

Dijkstra's algorithm: example

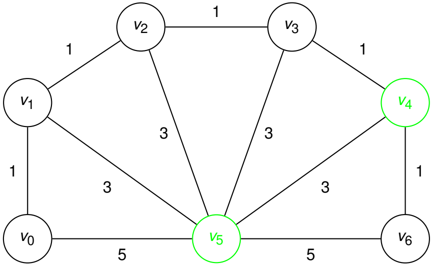
- Consider this example:



- Open list:
  v_3 v_5 v_4
- Closed list:
  $\{v_0 \mapsto 0, v_1 \mapsto 1, v_2 \mapsto 2, v_3 \mapsto 3, v_4 \mapsto 4, v_5 \mapsto 4\}$
- Step 5: Vertex v_2 inserted in front of v_5
- Done; found shortest distances to all vertices

Dijkstra's algorithm: example

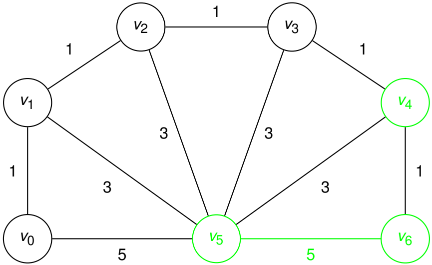
- Consider this example:



- Open list:
  v_5 v_4
- Closed list:
  $\{v_0 \mapsto 0, v_1 \mapsto 1, v_2 \mapsto 2, v_3 \mapsto 3, v_4 \mapsto 4, v_5 \mapsto 4\}$
- Step 5: Vertex v_2 inserted in front of v_5
- Done; found shortest distances to all vertices

Dijkstra's algorithm: example

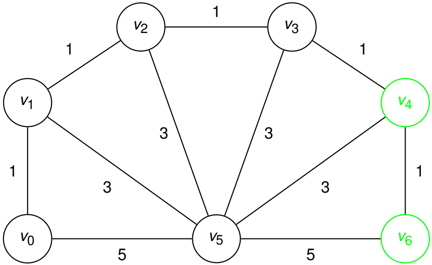
- Consider this example:



- Open list:
  v_5 v_4 v_6
- Closed list:
  $\{v_0 \mapsto 0, v_1 \mapsto 1, v_2 \mapsto 2, v_3 \mapsto 3, v_4 \mapsto 4, v_5 \mapsto 4, v_6 \mapsto 9\}$
- Step 5: Vertex v_2 inserted in front of v_5
- Done; found shortest distances to all vertices

Dijkstra's algorithm: example

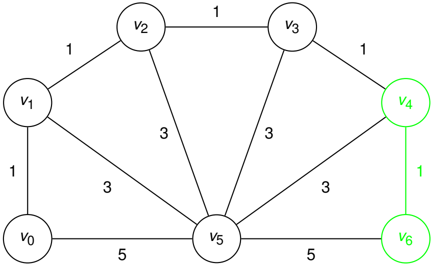
- Consider this example:



- Open list:
  v_4 v_6
- Closed list:
  $\{v_0 \mapsto 0, v_1 \mapsto 1, v_2 \mapsto 2, v_3 \mapsto 3, v_4 \mapsto 4, v_5 \mapsto 4, v_6 \mapsto 9\}$
- Step 5: Vertex v_2 inserted in front of v_5
- Done; found shortest distances to all vertices

Dijkstra's algorithm: example

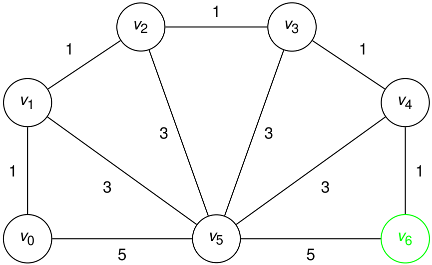
- Consider this example:



- Open list:
  v_4 v_6
- Closed list:
  $\{v_0 \mapsto 0, v_1 \mapsto 1, v_2 \mapsto 2, v_3 \mapsto 3, v_4 \mapsto 4, v_5 \mapsto 4, v_6 \mapsto 5\}$
- Step 5: Vertex v_2 inserted in front of v_5
- Done; found shortest distances to all vertices

Dijkstra's algorithm: example

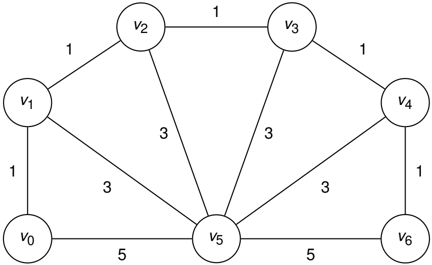
- Consider this example:



- Open list:
  v_6
- Closed list:
  $\{v_0 \mapsto 0, v_1 \mapsto 1, v_2 \mapsto 2, v_3 \mapsto 3, v_4 \mapsto 4, v_5 \mapsto 4, v_6 \mapsto 5\}$
- Step 5: Vertex v_2 inserted in front of v_5
- Done; found shortest distances to all vertices

Dijkstra's algorithm: example

- Consider this example:



- Open list:

- Closed list:
  $\{v_0 \mapsto 0, v_1 \mapsto 1, v_2 \mapsto 2, v_3 \mapsto 3, v_4 \mapsto 4, v_5 \mapsto 4, v_6 \mapsto 5\}$
- Step 5: Vertex v_2 inserted in front of v_5
- Done; found shortest distances to all vertices

Dijkstra's algorithm in a nutshell

- In a nutshell, Dijkstra's algorithm does the following:
  - For each vertex v, keep track of the shortest known distance $d(s, v)$ from s to v
    - Many versions of the algorithm initially set all distances to ∞ except s, where it is 0
    - Our **closed list** represents those vertices where we have found a non-∞ distance
  - In each iteration, expand a vertex v
    - This means exploring its outgoing edges (v, x)
    - v must be a previously unexpanded vertex with minimal $d(s, v)$
    - Our **open list** contains the candidates for this
  - During the expansion, update $d(s, x)$ if $d(s, v) + w(v, x)$ is smaller

Dijkstra’s algorithm: details

- We want to be able to output actual paths
- So we keep track of the **predecessor** of each vertex (i.e. from which other vertex we reached it, getting the shortest distance)
- Final output in the example:

Vertex	Distance	Shortest path
v_0	0	v_0
v_1	1	v_0, v_1
v_2	2	v_0, v_1, v_2
v_3	3	v_0, v_1, v_2, v_3
v_4	4	v_0, v_1, v_2, v_3, v_4
v_5	4	v_0, v_1, v_5
v_6	5	$v_0, v_1, v_2, v_3, v_4, v_6$

Dijkstra’s algorithm: implementation

- We make a few changes to our existing code:
- We use a dedicated **`Vertex`** class
  - This allows us to store additional information in them, which will be helpful soon
- We use maps (**`java.util.HashMap`** or **`std::map`**)
  - In adjacency lists to keep track of weights

```
public class Vertex{
    public int id;

    public HashMap<Vertex, Integer> adjacencyList;

    /* ... */
}
```

  - In the closed list to keep track of shortest path lengths

Dijkstra's algorithm: implementation

- We make a few changes to our existing code:
- We use a wrapper class containing
  - A vertex
  - The length of the shortest known path from *s* to *v*

```
public class WrappedVertex{
    public Vertex vertex;

    // shortest currently known distance from the start
    public int distance;

    public WrappedVertex(Vertex v, int d){
        vertex = v;
        distance = d;
    }
}
```

in order to store vertices in a heap

Dijkstra's algorithm: implementation

- We adapt the MinHeap class
  - It now stores wrapped vertices
  - It uses the current distance to figure out how far to sift up a new entry

```
public class MinHeap {
    ArrayList<WrappedVertex> items;

    public MinHeap(){
        items = new ArrayList<WrappedVertex>();
    }

    public void insert(Vertex v, int d){
        WrappedVertex newItem = new WrappedVertex(v, d);
        int index = items.size();
        items.add(newItem);
        // Determine position for insertion
        while(index > 0){
            int parent = (index-1)/2;  // parent index
            if(items.get(parent).distance > newItem.distance){ // sift up
                items.set(index, items.get(parent));
                index = parent;
            }
            else
                break;
        }
        items.set(index, newItem);
    }

    /* ... */
}
```

Dijkstra's algorithm: implementation

- Finally, we adapt the search method we have seen before to use a heap as its open list
- Note that if we find a new, shorter path to a vertex v, we
  - Create a new WrappedVertex with the new distance
  - Insert it in the open list
  - We may have several wrapped versions of the same vertex
    - They will have different distances
    - The one with the smallest distance will be handled first
    - The others will just be discarded eventually

Dijkstra’s algorithm: implementation

- The structure of the main loop looks like this:

```
public void Dijkstra(Vertex source){
   // Set up open, closed, and predecessor lists
   MinHeap o = new MinHeap();
   HashMap<Vertex, Integer> c = new HashMap<>();
   HashMap<Vertex, Vertex> p = new HashMap<>();
   o.insert(source, 0);
   c.put(source, 0);
   p.put(source, null);
   while(!o.empty()){
      WrappedVertex w = o.extractMinimum();
      if(w.distance == c.get(w.vertex)){  // w is up to date
         /* TO DO: complete this
             Output the distance and shortest path (follow the chain of predecessors)
             For every edge e out of w.vertex,
               - check if e gives a new shortest path to its target
               - update lists if yes
         */
      }
   }
}
```

Dijkstra's algorithm: analysis

- Dijkstra's algorithm is an example of a **greedy** algorithm
  - It always picks the option that currently looks best
- It expands each vertex (i.e. examines its outgoing edges) **once**
- Its complexity depends on details such as graph representation, sparsity, etc.
  A general lower bound is $\Theta(|E| + |V|^2)$.