CM2035 Topic 10 Graph Algorithms

Ian Sanders

October 2024 Session
April 2025 Session

UNIVERSITY OF LONDON

Recap

In the module videos we have seen

- Ways of representing graphs.
- Minimum Spanning Tree (MST) algorithms (Prim's and Kruskal's).
- Shortest Path algorithms (Djikstra's)

A different algorithm

In this webinar we are going to look at a slightly different graph algorithm.

A different algorithm

In this webinar we are going to look at a slightly different graph algorithm.

An algorithm to determine whether there is a path between two nodes in the graph

A different algorithm

In this webinar we are going to look at a slightly different graph algorithm.

An algorithm to determine whether there is a path between two nodes in the graph

We will do this for two reasons:

1. To reinforce the ideas behind dealing with graphs.

A different algorithm

In this webinar we are going to look at a slightly different graph algorithm.

An algorithm to determine whether there is a path between two nodes in the graph

We will do this for two reasons:

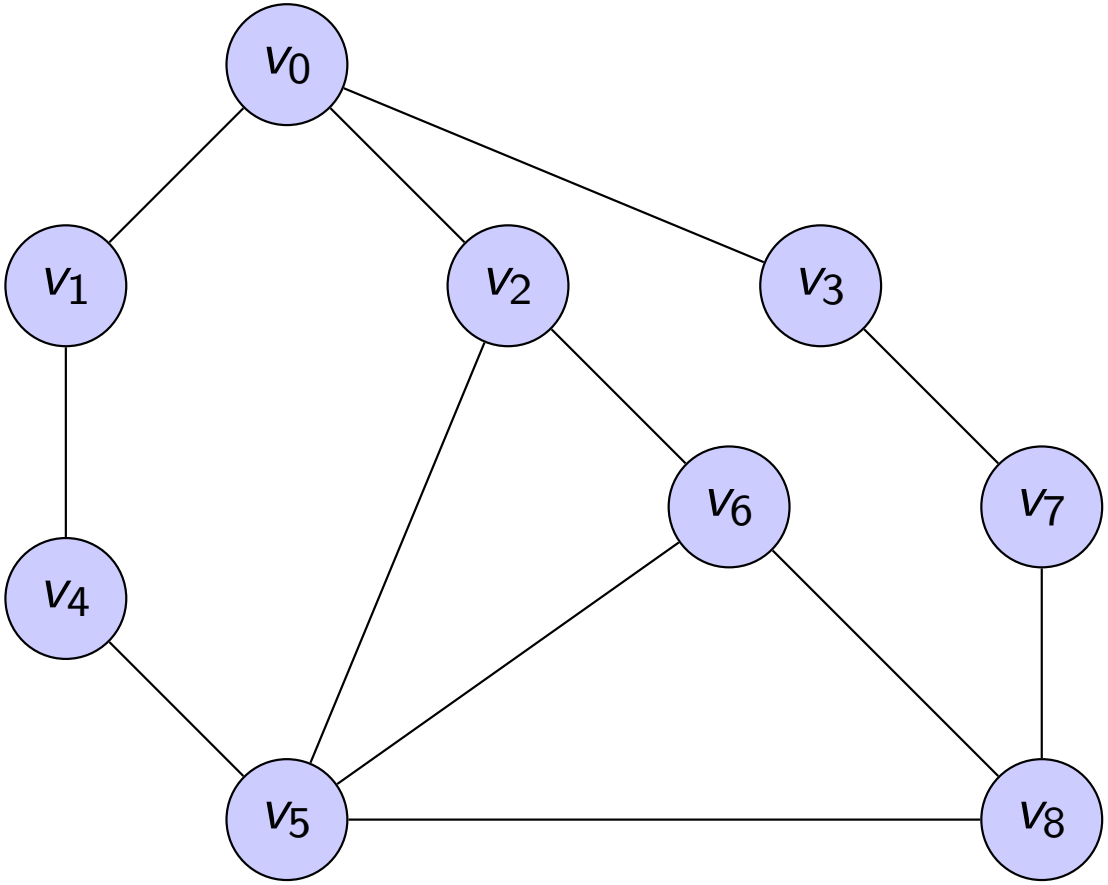
1. To reinforce the ideas behind dealing with graphs.
2. To relook at depth first search (and breadth first search) but on graphs rather than trees (as we considered before).

The question

Definition: A path is a sequence of non-repeated nodes $v_m, v_l, \ldots, v_k$ connected through edges present in a graph

Question: Given a graph $G(V, E)$ is there a path between v_i and v_j in G?

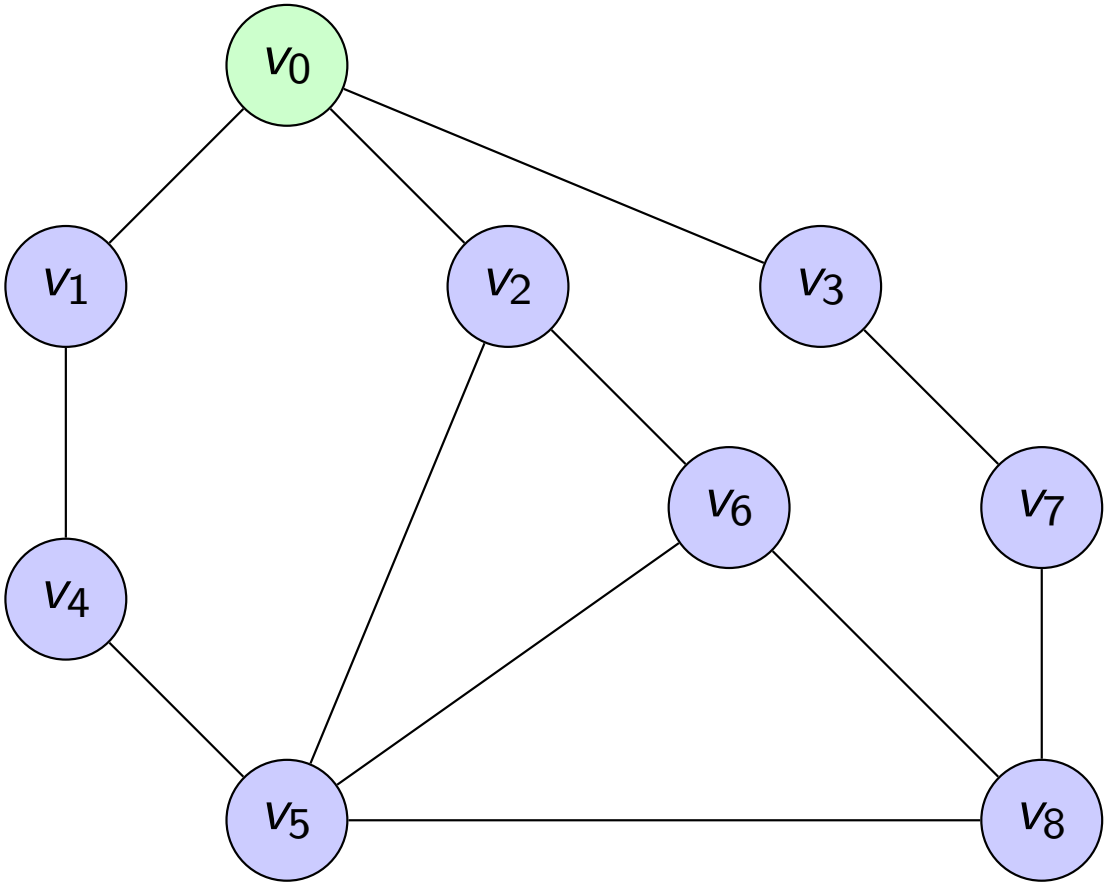
Example



Suppose we want to find a path between v_0 and v_4.

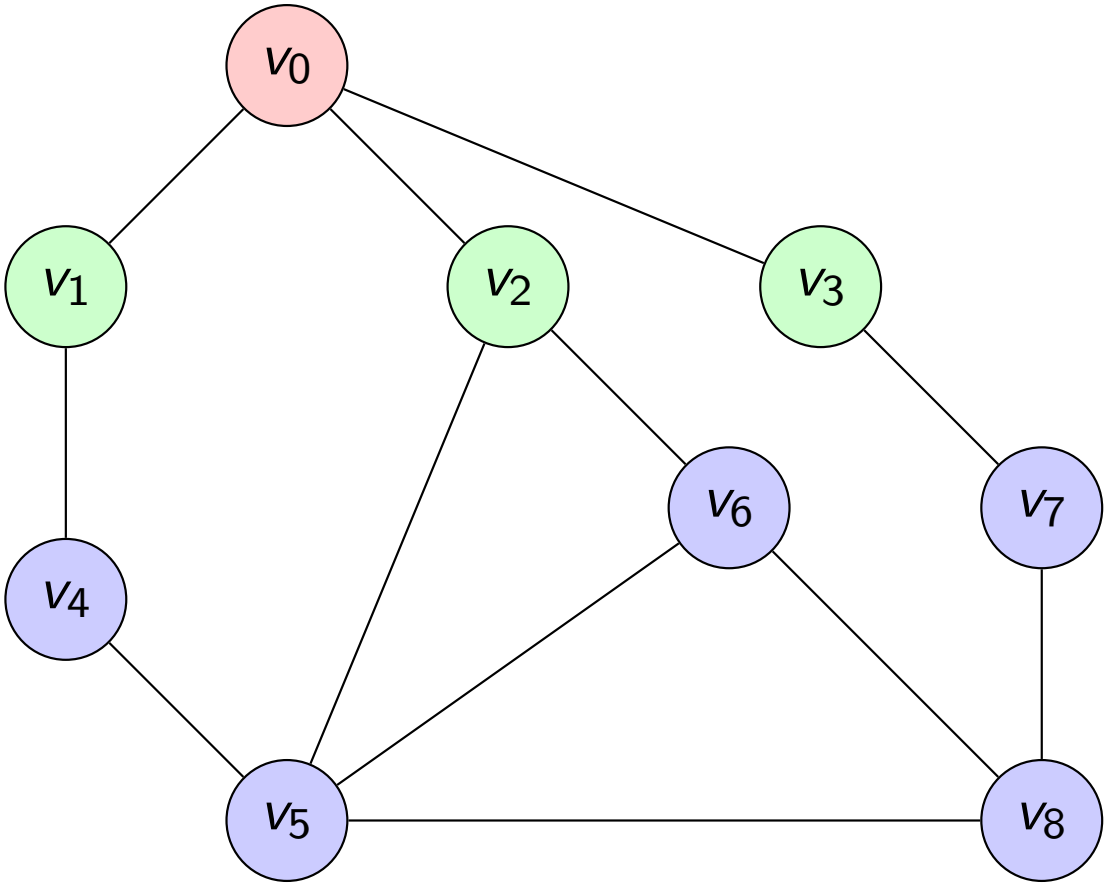
Suppose we want to find a path between v_0 and v_4.
We use a stack called `Frontier` (green nodes) and a set called `explored` (red nodes) to keep track of where we are in our search.

Suppose we want to find a path between v_0 and v_4.
We use a stack called `Frontier` (green nodes) and a set called `explored` (red nodes) to keep track of where we are in our search.
We start by making the `Frontier` our start node and setting `explored` to be empty.



Frontier $= \{v_0\}$ and explored $= \{\}$

The first node in the `Frontier` is not our goal so we add it to `explored` and then *expand* it.
Here *expanding* means finding its children and pushing them onto `Frontier` as we find them provided they have not already been explored and are not already in `Frontier`.



`Frontier` $= \{v_3, v_2, v_1\}$ and `explored` $= \{v_0\}$

We continue our search by

Checking if `Frontier` is empty. If it is then return *failure*.

Pop the first node from the stack

If it is v_4 then we have found a path return *success*

If it is not then add that node to the explored list and expand that node

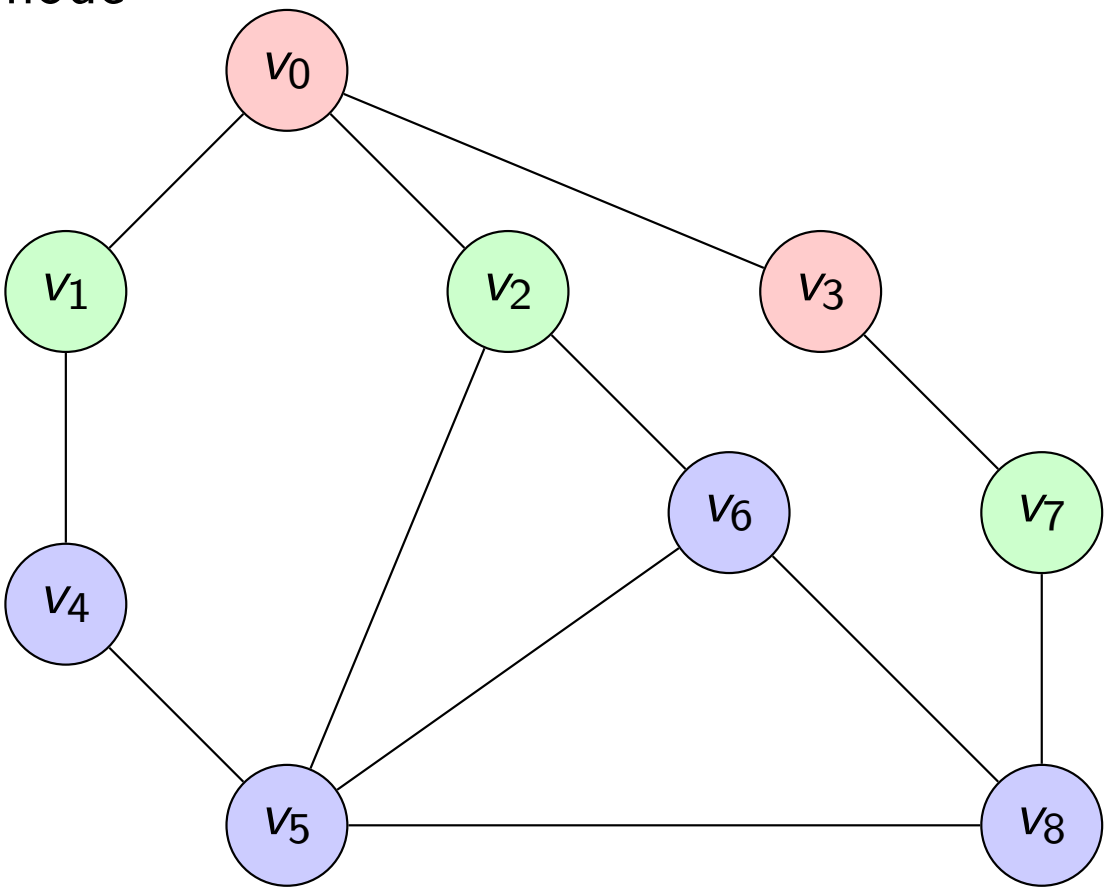
We continue our search by

Checking if `Frontier` is empty. If it is then return *failure*.

Pop the first node from the stack

If it is v_4 then we have found a path return *success*

If it is not then add that node to the explored list and expand that node



`Frontier` $= \{v_7, v_2, v_1\}$ and `explored` $= \{v_0, v_3\}$

We continue our search by

Checking if `Frontier` is empty. If it is then return *failure*.

Pop the first node from the stack

If it is v_4 then we have found a path return *success*

If it is not then add that node to the explored list and expand that node

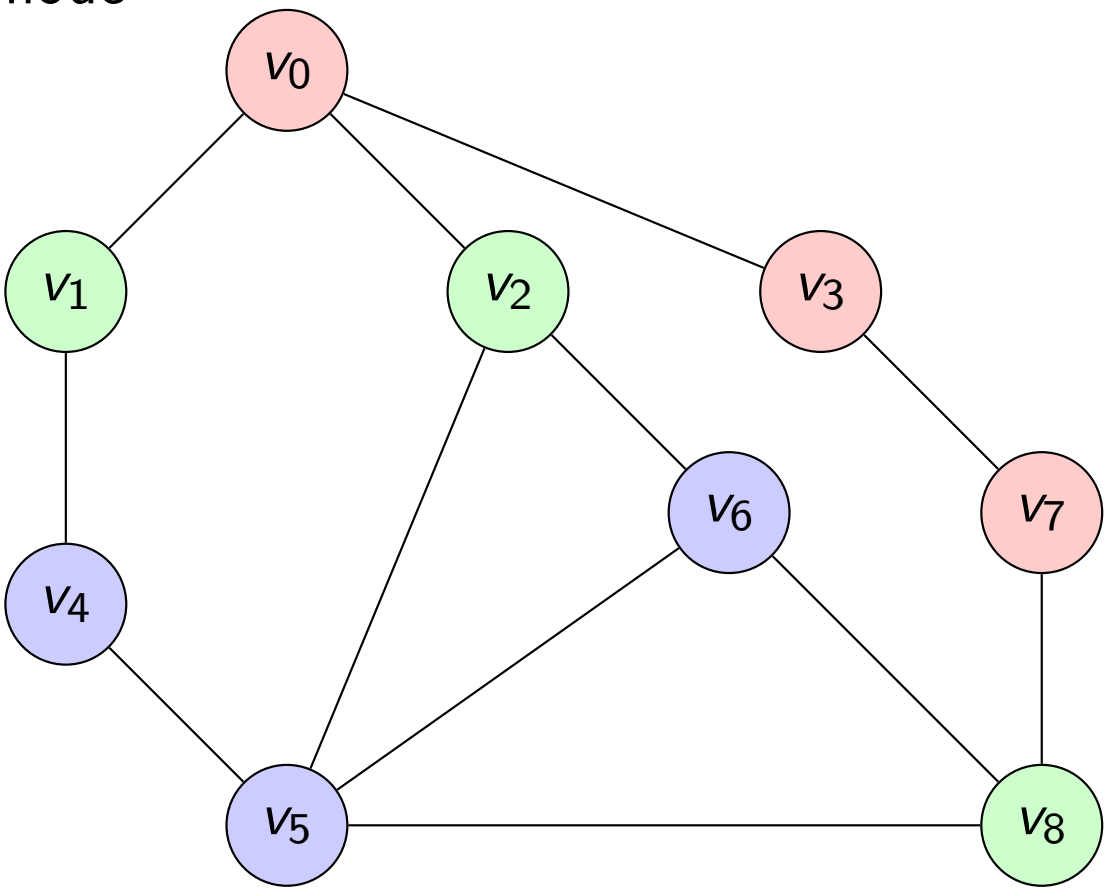
We continue our search by

Checking if `Frontier` is empty. If it is then return *failure*.

Pop the first node from the stack

If it is v_4 then we have found a path return *success*

If it is not then add that node to the explored list and expand that node



`Frontier` $= \{v_8, v_2, v_1\}$ and `explored` $= \{v_0, v_3, v_7\}$

We continue our search by

Checking if `Frontier` is empty. If it is then return *failure*.

Pop the first node from the stack

If it is v_4 then we have found a path return *success*

If it is not then add that node to the explored list and expand that node

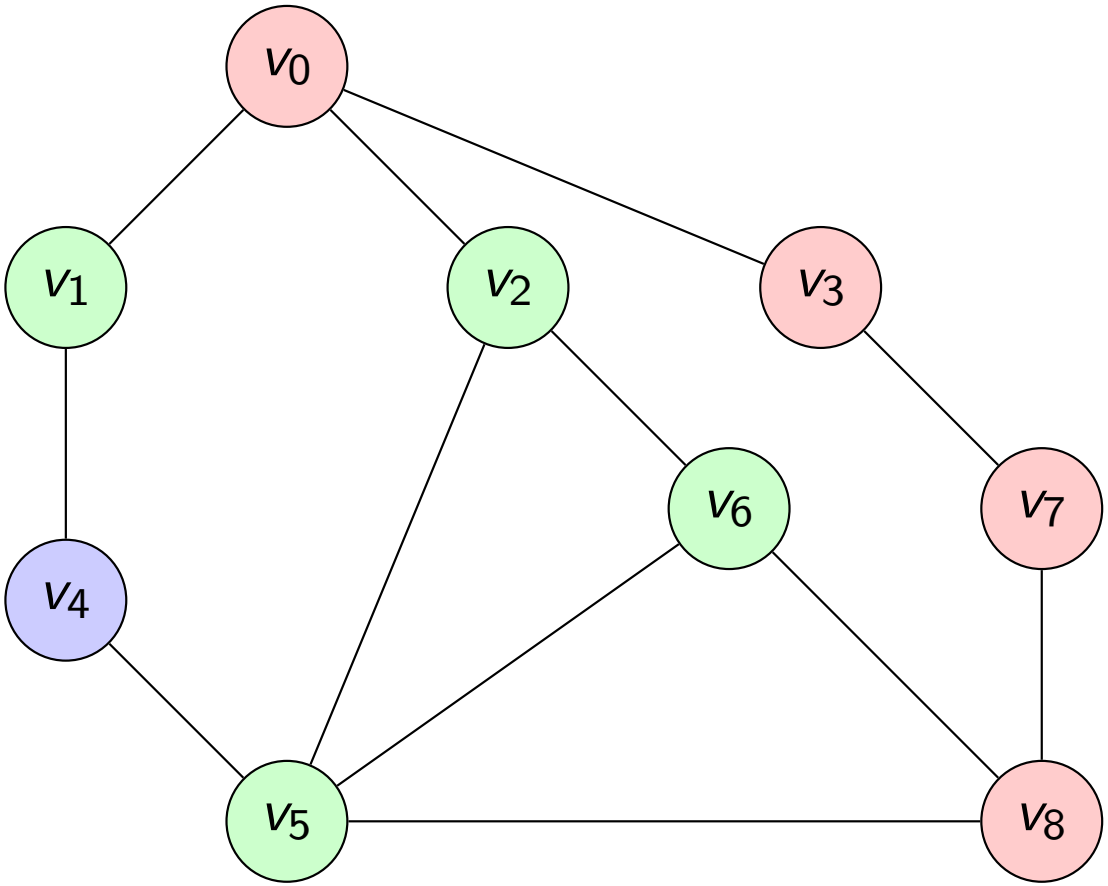
We continue our search by

Checking if `Frontier` is empty. If it is then return *failure*.

Pop the first node from the stack

If it is v_4 then we have found a path return *success*

If it is not then add that node to the explored list and expand that node



`Frontier` $= \{v_5, v_6, v_2, v_1\}$ and `explored` $= \{v_0, v_3, v_7, v_8\}$

We continue our search by

Checking if `Frontier` is empty. If it is then return *failure*.

Pop the first node from the stack

If it is v_4 then we have found a path return *success*

If it is not then add that node to the explored list and expand that node

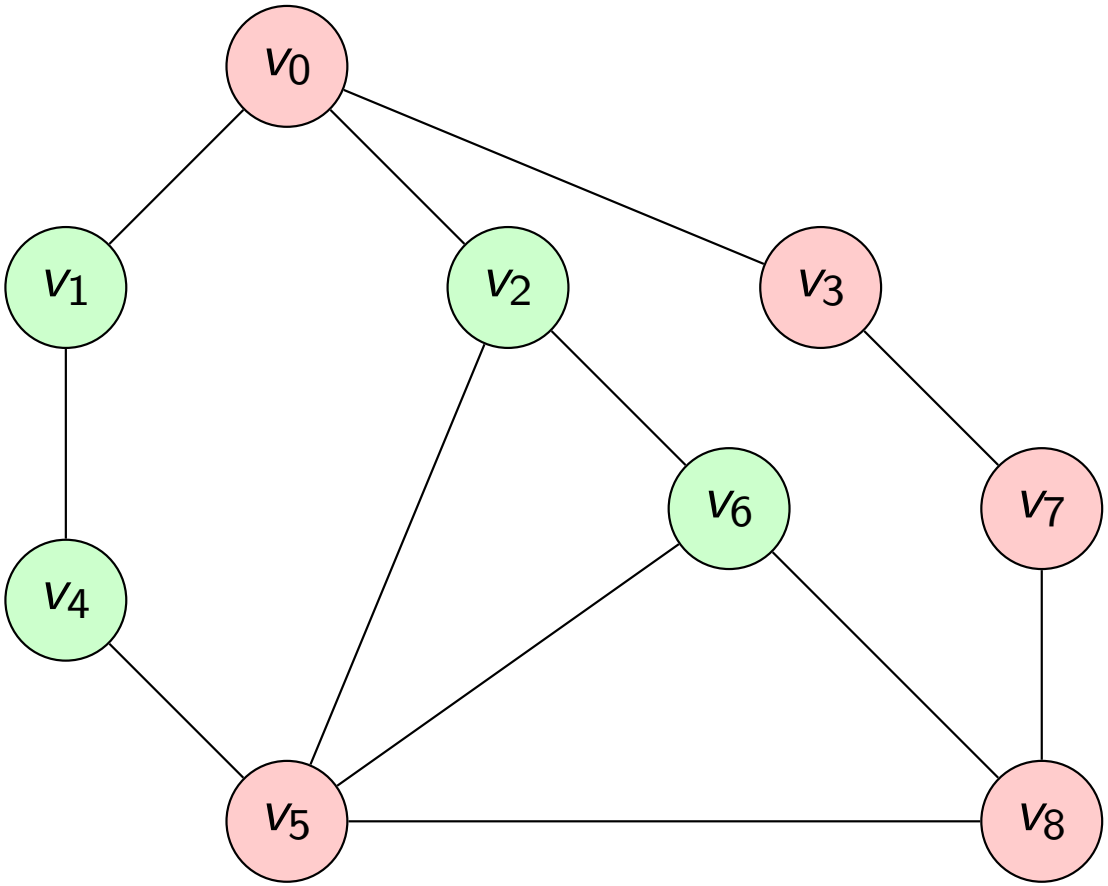
We continue our search by

Checking if `Frontier` is empty. If it is then return *failure*.

Pop the first node from the stack

If it is v_4 then we have found a path return *success*

If it is not then add that node to the explored list and expand that node



`Frontier` $= \{v_4, v_6, v_2, v_1\}$ and `explored` $= \{v_0, v_3, v_7, v_8, v_5\}$

We continue our search by

Checking if `Frontier` is empty. If it is then return *failure*.

Pop the first node from the stack

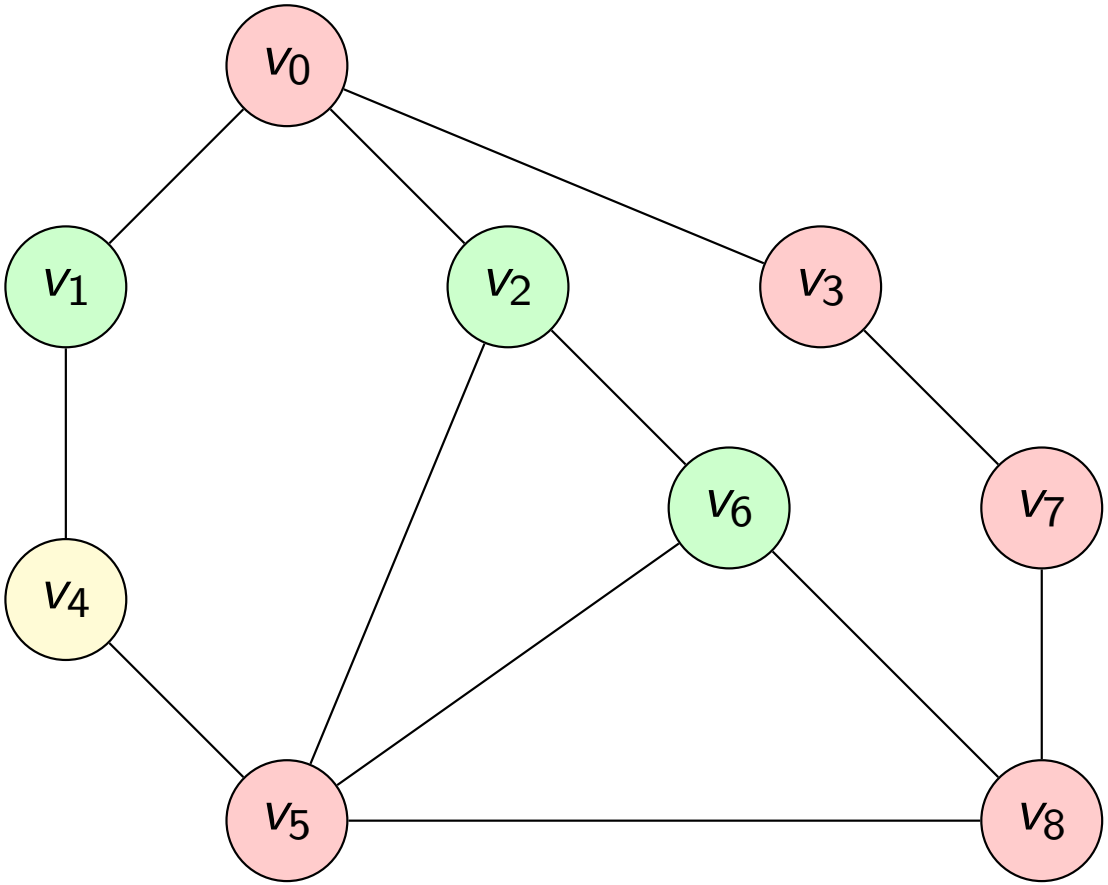
In this case it is v_4 so we have found a path. Return *success*

We continue our search by

Checking if `Frontier` is empty. If it is then return *failure*.

Pop the first node from the stack

In this case it is v_4 so we have found a path. Return *success*



`Frontier` $= \{v_4, v_6, v_2, v_1\}$ and `explored` $= \{v_0, v_3, v_7, v_8, v_5\}$

Notes

- The search as shown above is essentially a *depth first search* on a graph.

Notes

- The search as shown above is essentially a *depth first search* on a graph.
- The order in which nodes are expanded is governed by the order they are pushed onto and popped from the stack. This depends on the order in which they are encountered in the "edge list".

Notes

- The search as shown above is essentially a *depth first search* on a graph.
- The order in which nodes are expanded is governed by the order they are pushed onto and popped from the stack. This depends on the order in which they are encountered in the "edge list".
- The search could be changed to be a *breadth first search* by using a *queue* instead of a stack to store the frontier nodes.

```
00  Algorithm pathbetween(G(V, E), v_i, v_j)
01      Let frontier be an empty stack
02      Let explored be an empty set
03      Push v_i onto frontier
04      Loop:
05          If frontier is empty
06              Return False
07          Else
08              current ← pop(frontier)
09              If current = v_j
10                  Return True
11              Else
12                  explored ← explored + current
13                  For all neighbours n of current
14                      If n ∉ frontier and n ∉ explored
15                          push n onto frontier
```

Analysis

- In the worst case each vertex would have to be explored. This means going through the loop (from lines 04 to 15) for each vertex. This is $\Theta(|V|)$.

Analysis

- In the worst case each vertex would have to be explored. This means going through the loop (from lines 04 to 15) for each vertex. This is $\Theta(|V|)$.
- Inside the loop there is a `for` loop to deal with all of the neighbours of the vertex currently being explored.

Analysis

- In the worst case each vertex would have to be explored. This means going through the loop (from lines 04 to 15) for each vertex. This is $\Theta(|V|)$.
- Inside the loop there is a `for` loop to deal with all of the neighbours of the vertex currently being explored.
- In the worst case (and depending on the graph representation used) each edge in E would need to be considered to find the edges leaving the current vertex and to determine the neighbours. This would be $\Theta(|E|)$.

Analysis

- In the worst case each vertex would have to be explored. This means going through the loop (from lines 04 to 15) for each vertex. This is $\Theta(|V|)$.
- Inside the loop there is a `for` loop to deal with all of the neighbours of the vertex currently being explored.
- In the worst case (and depending on the graph representation used) each edge in E would need to be considered to find the edges leaving the current vertex and to determine the neighbours. This would be $\Theta(|E|)$.
  The algorithm would thus be $\Theta(|V||E|)$ in the worst case.

Analysis

- In the worst case each vertex would have to be explored. This means going through the loop (from lines 04 to 15) for each vertex. This is $\Theta(|V|)$.
- Inside the loop there is a `for` loop to deal with all of the neighbours of the vertex currently being explored.
- In the worst case (and depending on the graph representation used) each edge in E would need to be considered to find the edges leaving the current vertex and to determine the neighbours. This would be $\Theta(|E|)$.
  The algorithm would thus be $\Theta(|V||E|)$ in the worst case.
- If an *adjacency list* representation is used to store the edges then the neighbours for each can be found directly and thus only $|E|$ edges in total need to be considered.

Analysis

- In the worst case each vertex would have to be explored. This means going through the loop (from lines 04 to 15) for each vertex. This is $\Theta(|V|)$.
- Inside the loop there is a `for` loop to deal with all of the neighbours of the vertex currently being explored.
- In the worst case (and depending on the graph representation used) each edge in E would need to be considered to find the edges leaving the current vertex and to determine the neighbours. This would be $\Theta(|E|)$.
  The algorithm would thus be $\Theta(|V||E|)$ in the worst case.
- If an *adjacency list* representation is used to store the edges then the neighbours for each can be found directly and thus only $|E|$ edges in total need to be considered.
  So the algorithm would then be $\Theta(|V| + |E|)$ in the worst case

Good Luck with the exam

Post questions on the discussion forum if you have any.

UNIVERSITY OF LONDON