Greedy Algorithms 1

Horses for Courses

Different problem solving strategies suite different kinds of problems

- Greedy method
- Dynamic Programming
- Network Flow

Five Representative Problems

- Different strategies suitable for different problems
- Consider 5 example problems that exhibit some of the possibilities
  - interval scheduling
  - weighted interval scheduling
  - bipartite matching
  - independent set
  - competitive facility location

Problem 1: The Interval Scheduling Problem

The idea:

- Some resource is available
- Set of intervals that can be scheduled
- Each interval has fixed start and end time
- Goal is to schedule as many of the intervals as possible
- Resource can only be applied to one interval at a time

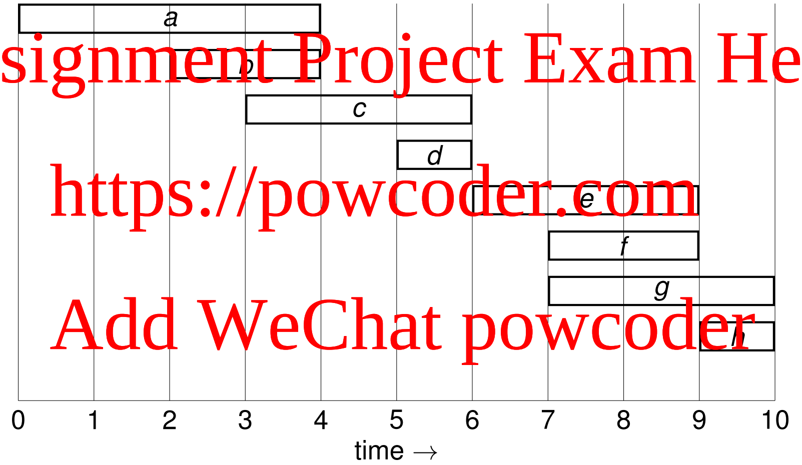
Specifying the Problem

Two intervals are **compatible** if they have non-overlapping intervals

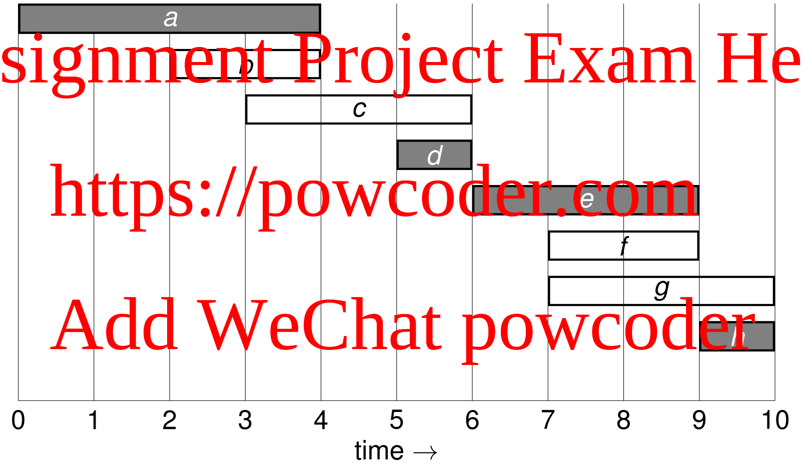
Input: A set of intervals where each interval has a start and end time

Output: The largest selection of compatible intervals that can be made

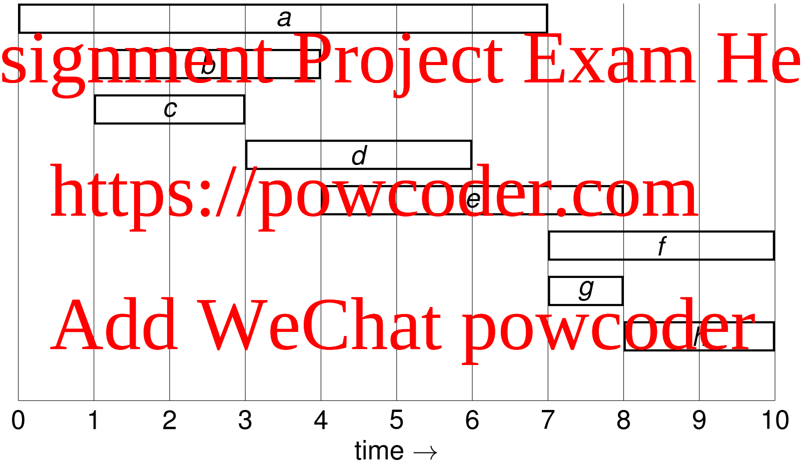
Interval Scheduling Example



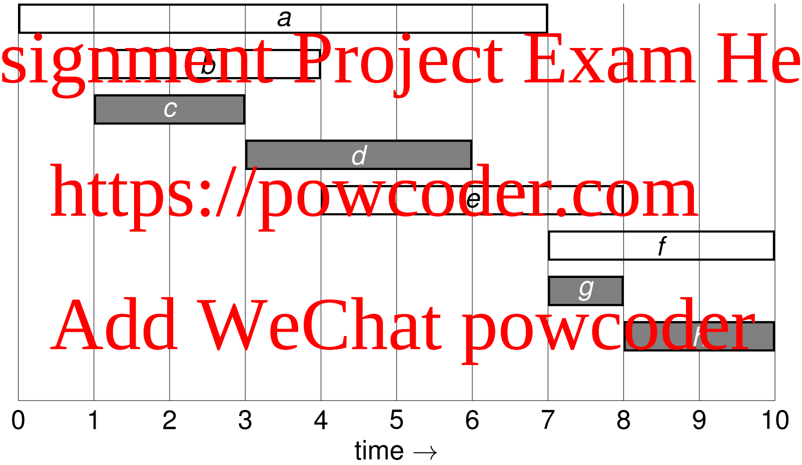
Interval Scheduling Example



One for you to solve



One for you to solve



Greedy Interval Scheduling

Can be solved using the so-called **greedy** approach

- make series of locally best choices
- very efficient way to solve problems
- be greedy when you can get away with it
- can't always get away with it
- some problems not amenable to greedy approach
- to be continued...

Problem 2: Weighted Interval Scheduling

A variant of the Interval Scheduling Problem

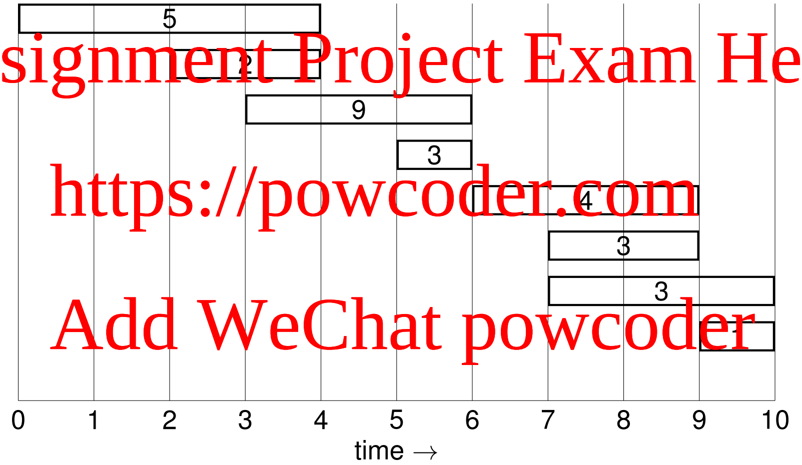
Each interval has been assigned a **weight**

This could be a measure of how important it is that the interval is scheduled

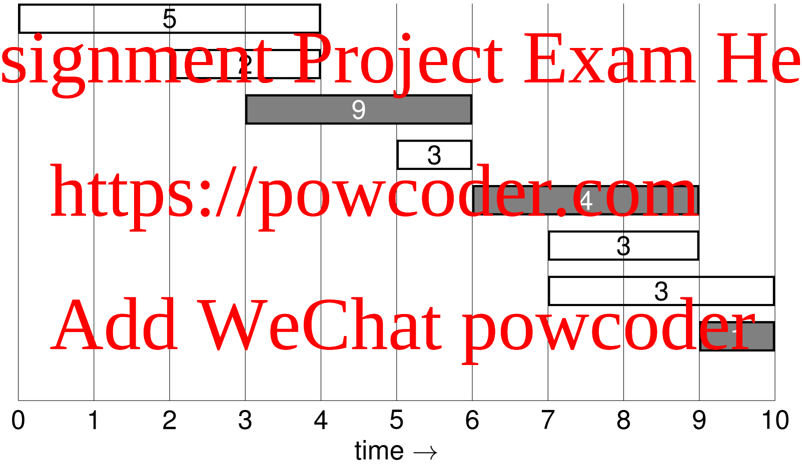
Input: set of intervals with start and finish times plus weights
Output: subset of compatible intervals with greatest total weight

Note that the number of intervals chosen is not important

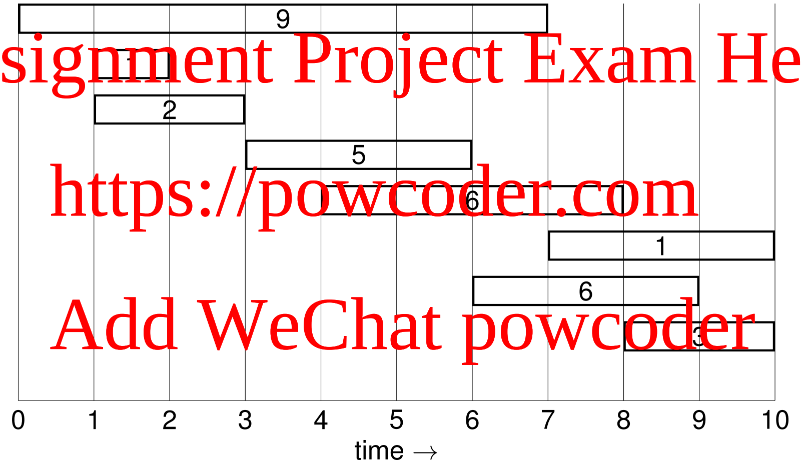
Weighted Interval Scheduling Example



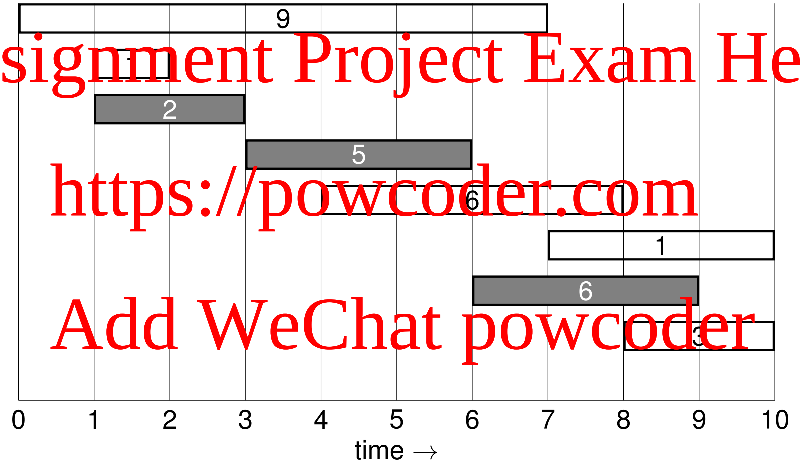
Weighted Interval Scheduling Example



One for you to solve



One for you to solve



Solving Weighted Interval Scheduling

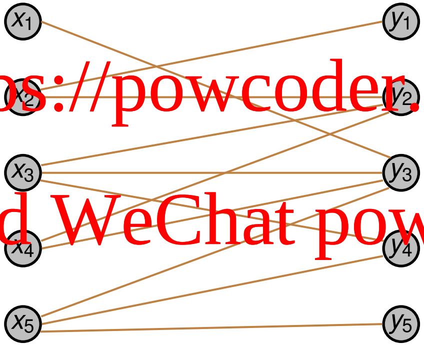
Cannot be solved using the greedy approach

Requires the more powerful technique of **dynamic programming**

- systematically decompose problems into subproblems
- based on **divide-and-conquer** approach
- solutions to subproblems recorded in table
- to be continued. . .

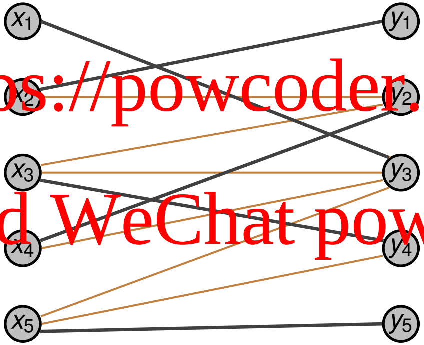
Bipartite Matching

Input: a graph that is bipartite (leave details of definition for later)
Output: matching with most edges possible



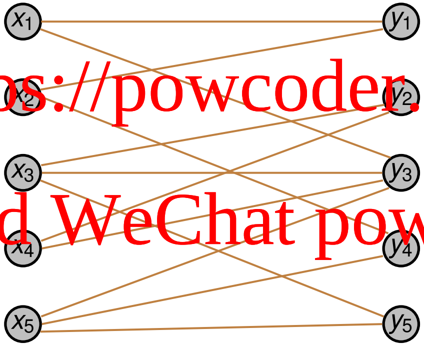
Bipartite Matching

Input: a graph that is bipartite (leave details of definition for later)
Output: matching with most edges possible



One for you to solve

Input: a graph that is bipartite (leave details of definition for later)
Output: matching with most edges possible



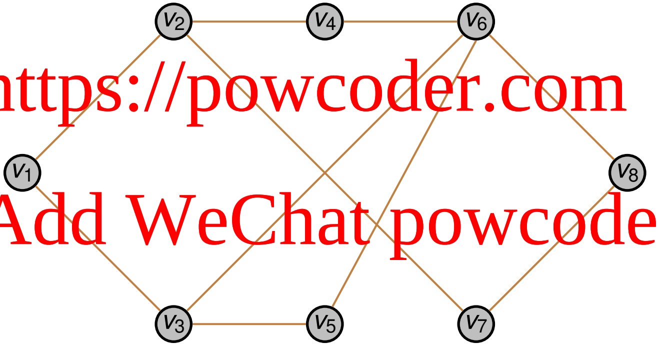
Bipartite Matching

Can be solved using the so-called **network flow** approach

- Bipartite matching problem can be seen as special case of network flow problem
- Network flow problem widely applicable
- More later...

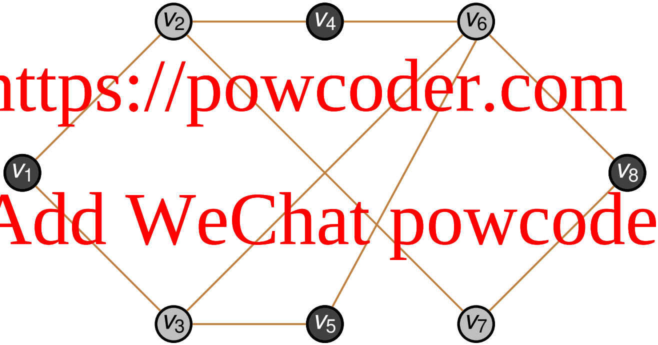
Independent Set

Input: a graph
Output: largest independent set: no two nodes joined by edge

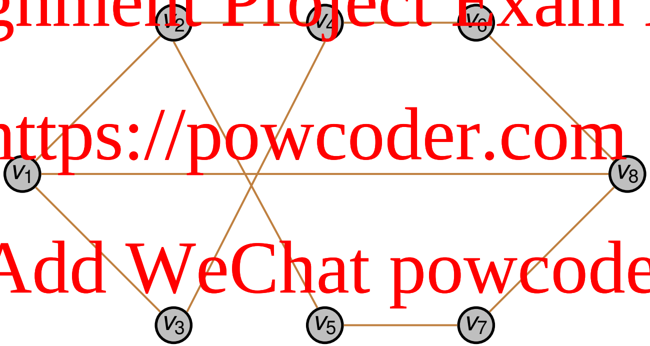


Independent Set

Input: a graph
Output: largest independent set: no two nodes joined by edge



One for you to solve



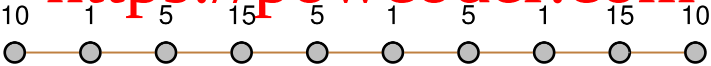
Independent Set

NP-complete problem

- no efficient algorithm known that solves this problem
- exponential number of subsets to consider

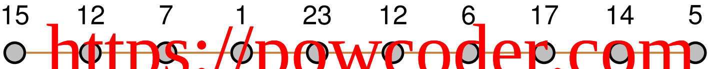
Competitive Facility Location

Input: A graph with weighted *nodes*
Output: Players alternate choosing nodes, but can't select node when neighbouring node has already been chosen
Goal is to maximise weight of selected nodes



Second player can't exceed total node weight of 20

One for you to solve



Competitive Facility Location

P-SPACE-complete problem

- Even harder than NP-complete
- Compare difficulty in verifying that solution is valid
- Seems to requires polynomial space

Greedy Algorithms:
The Interval Scheduling Problem

Summary

- What makes an algorithm a greedy algorithm
- Interval Scheduling Problem
- Shortest Path Problem
- Minimum Spanning Tree Problem
- Construction of Huffman Codes

The Greedy Approach

An approximate characterisation:

- The algorithm makes a series of choices
- A solution is built up as these choices are made
- The **locally** best option is made at each stage

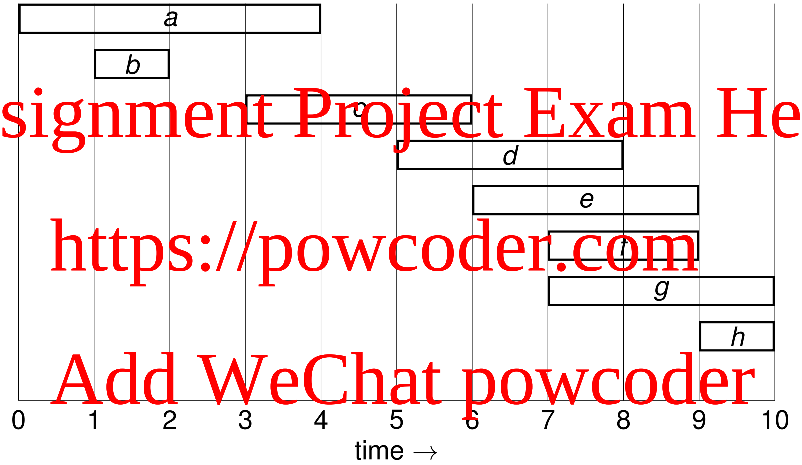
When is this approach appropriate?

When Greed Works

- No need to worry about what will happen in the future
- Safe to commit to each decision
- No backtracking on decisions
- Guarantee that the **globally** best solution will be found

Some problems are amenable to this approach while others aren't

Interval Scheduling Problem



Problem: to schedule as many of the intervals as possible

The Series of Choices

Setting up the approach:

- Working from time 0 forward in time
- Decisions involving choosing which of the intervals to schedule next

Question: On what basis can the choice be made?

Possible Strategy: Choose the interval that can be started earliest

- Does this guarantee that an optimal solution is found?

A Non-optimal Strategy

- Choosing first available interval is a bad strategy
- A very long interval could be chosen
- See interval *a* in previous example
- Could be better to choose a shorter interval that ends earlier
- In this case better to choose interval *b*
- Problem with this strategy is we are only counting total number of intervals scheduled

A Better Strategy

- Among those intervals that could be scheduled next, choose the interval with the earliest possible **ending** point
- An interval can be scheduled if the start time has not yet passed
- Use a priority queue to hold the intervals

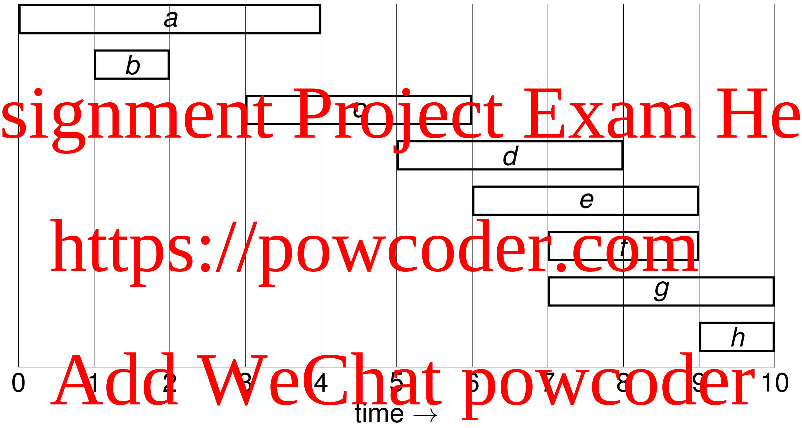
- Use the interval's end time as its priority
- Earlier end times have higher priority than later ones
- For an interval α let $s(\alpha)$ and $f(\alpha)$ be the start and finish times of α, respectively

Interval Scheduling Algorithm

IntervalSchedule(I) :

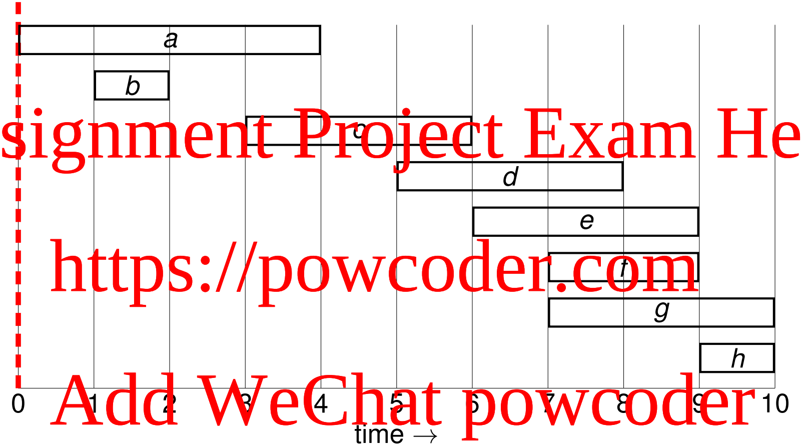
let Q *be queue of intervals prioritised by ealiest end time*
initialise A*, the set of scheduled intervals, to be the empty set*
initialise current time k *to be* 0
while Q *is not yet empty*
    remove interval α *from the front of* Q
    if $k \leq s(\alpha)$ *then*
        add the interval α *to the set* A
        set the current time k *to be* $f(\alpha)$
return A

Illustrating the Interval Scheduling Algorithm



First create priority queue Q, initialise A and set current time to 0

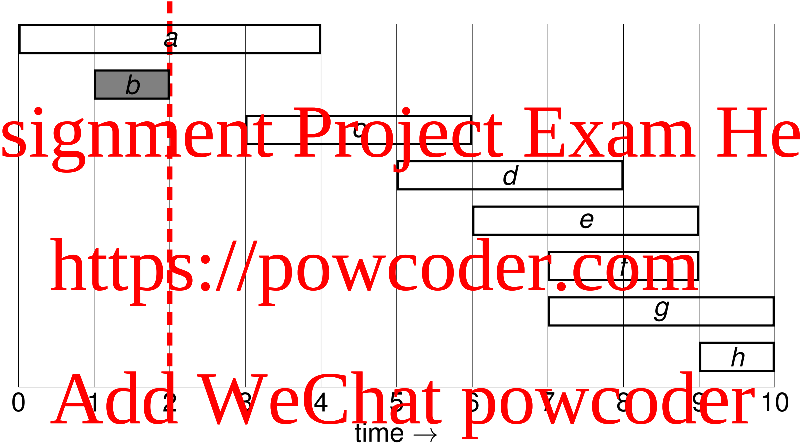
Illustrating the Interval Scheduling Algorithm



$Q = (b, a, c, d, e, f, g, h)$ $A = \{\}$

Remove b from Q; b can be scheduled so add b to A and reset time

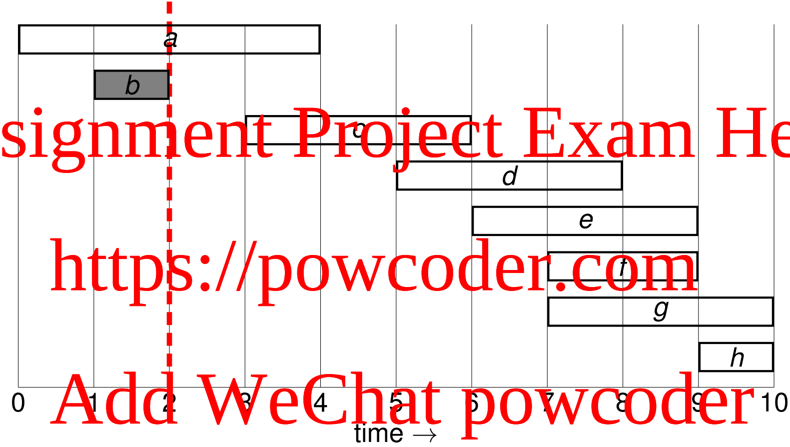
Illustrating the Interval Scheduling Algorithm



$Q = (a, c, d, e, f, g, h)$ $A = \{b\}$

Remove *a* from *Q*, but too late to schedule *a*

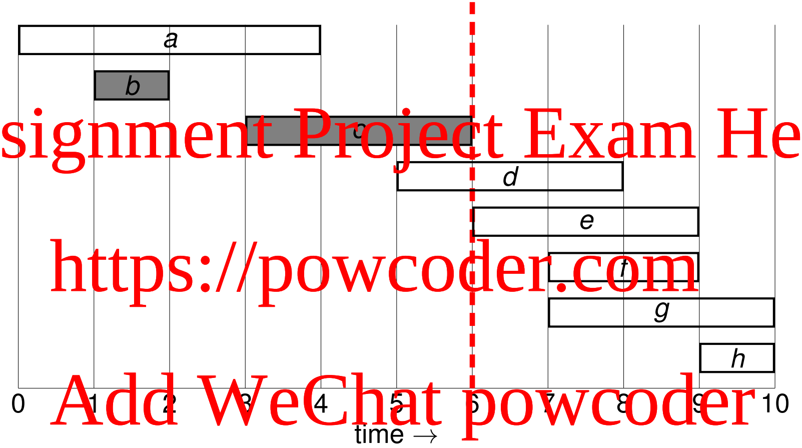
Illustrating the Interval Scheduling Algorithm



$Q = (c, d, e, f, g, h)$ $\qquad$ $A = \{b\}$

Remove c from Q; c can be scheduled so add c to A and reset time

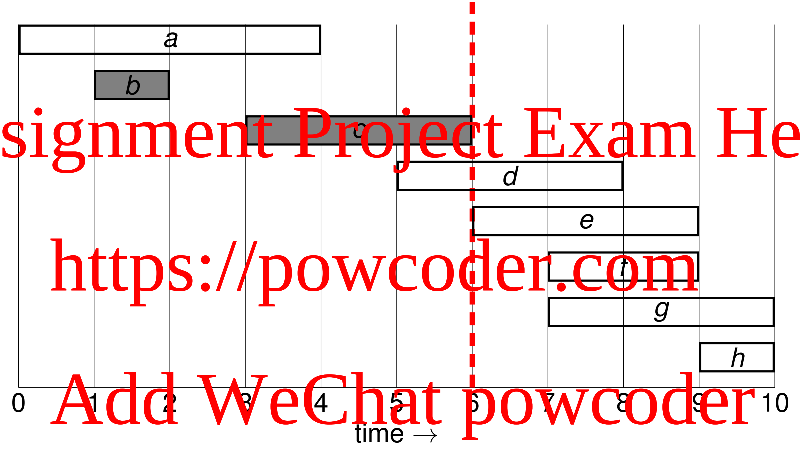
Illustrating the Interval Scheduling Algorithm



$Q = (d, e, f, g, h)$ $A = \{b, c\}$

Remove d from Q, but too late to schedule d

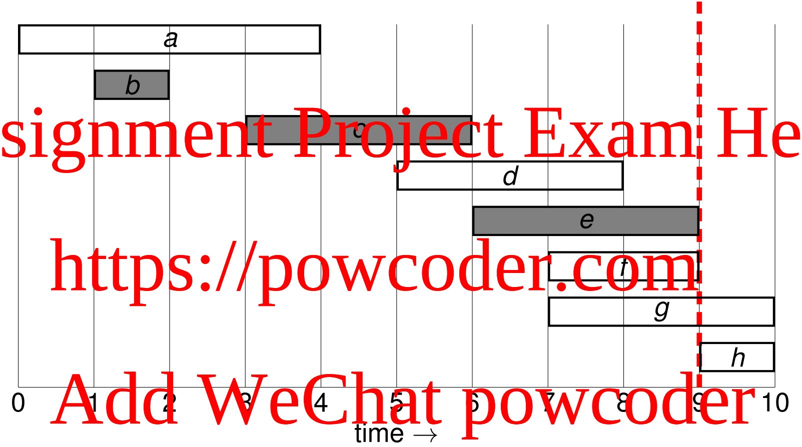
Illustrating the Interval Scheduling Algorithm



$Q = (e, f, g, h)$ $A = \{b, c\}$

Remove *e* from *Q*; *e* can be scheduled so add *e* to *A* and reset time

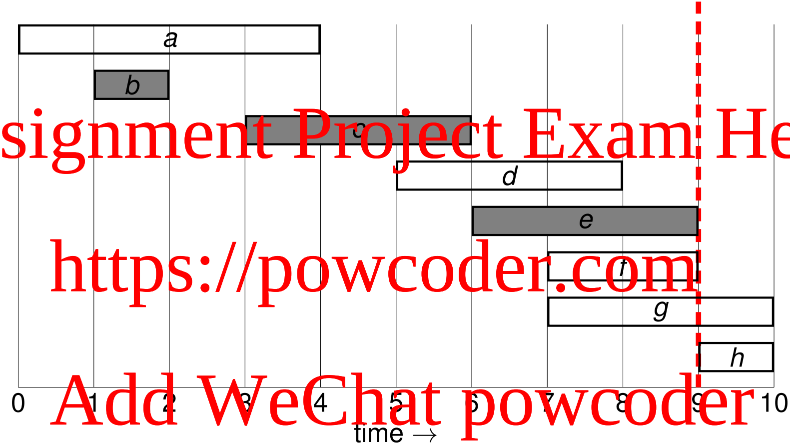
Illustrating the Interval Scheduling Algorithm



$Q = (f, g, h)$ $A = \{b, c, e\}$

Remove f from Q, but too late to schedule f

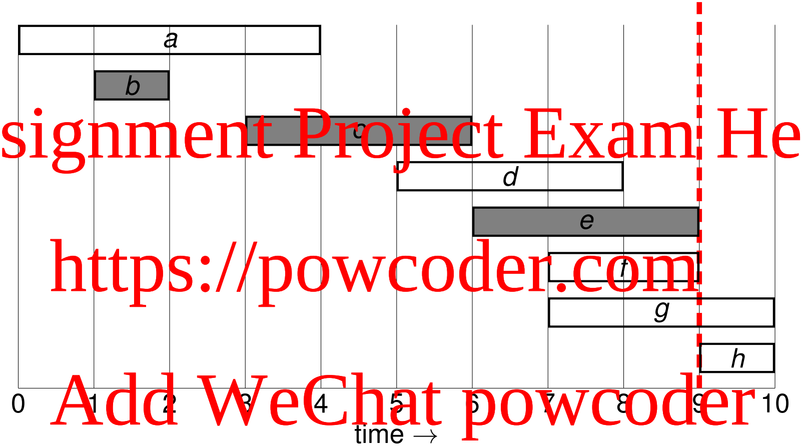
Illustrating the Interval Scheduling Algorithm



$Q = (g, h)$ $A = \{b, c, e\}$

Remove g from Q, but too late to schedule g

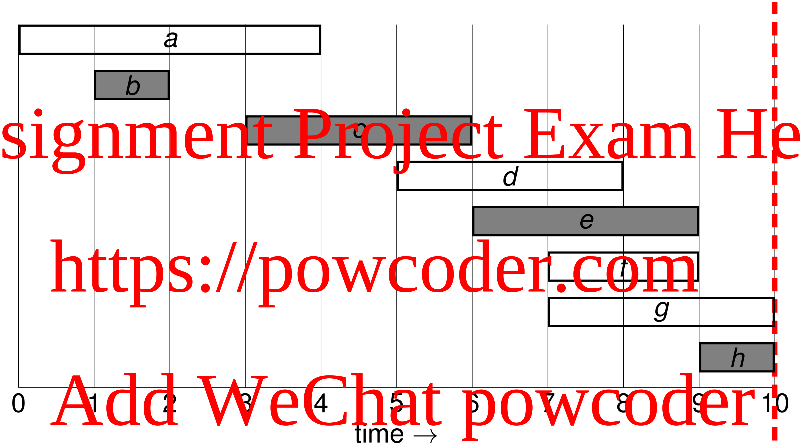
Illustrating the Interval Scheduling Algorithm



$Q = (h)$ $A = \{b, c, e\}$

Remove h from Q; h can be scheduled so add h to A and reset time

Illustrating the Interval Scheduling Algorithm



$Q = ()$ $A = \{b, c, e, h\}$

Q is empty so all done!

Correctness of Algorithm

All intervals in A are compatible

- The intervals do not overlap
- This is guaranteed by the fact that we check that the start time of each interval is not before the end time of the one previously scheduled

Correctness of Algorithm (cont.)

We will show that for any optimal solution B we know that $|A| = |B|$

- A is as good as any optimal solution
- We will show that the solution A **keeps ahead** of any optimal solution B.

Consider some optimal solution B

- Let $(\alpha_1, \ldots, \alpha_k)$ be the elements of A in scheduled order
- Let $(\beta_1, \ldots, \beta_m)$ be the elements of B in scheduled order

Correctness of Algorithm (cont.)

Claim: A stays ahead of B

$$f(\alpha_i) \leq f(\beta_i) \quad \text{for all } i \text{ such that } 1 \leq i \leq k$$

Proof by induction on i:

Basis of induction: clearly true for $i = 1$

Inductive step:

- intervals in B are compatible so $f(\beta_{i-1}) \leq s(\beta_i)$
- by induction $f(\alpha_{i-1}) \leq f(\beta_{i-1})$
- so $f(\alpha_{i-1}) \leq s(\beta_i)$
- so β_i was available to be scheduled when α_i was
- so $f(\alpha_i) \leq f(\beta_i)$

Correctness of Algorithm (cont.)

Claim: A is as good as B

Proof by contradiction

- Suppose B is better than A
- Must be more elements in B than A, so $m > k$
- Because A stays ahead of B we know that $f(\alpha_k) \leq f(\beta_k)$
- So the algorithm would have included β_{k+1} in A
- Since β_{k+1} is compatible with β_k it must be compatible with α_k

Interval Scheduling Algorithm

IntervalSchedule(I) :

let Q *be the priority queue containing all intervals in* I
initialise A*, the set of scheduled intervals, to be the empty set*
initialise current time k *to be* 0
while Q *is not yet empty*
    remove interval α *from the front of* Q
    if $k \leq s(\alpha)$ *then*
        add the interval α *to the set* A
        set the current time k *to be* $f(\alpha)$
return A

Analysis of Running Time

- What is a good measure of progress
- Number of intervals that have been considered
- Always increases by one each time loop is executed
- Limits number of executions of loop to n
- Suppose priority queue implemented as a heap
- $O(n)$ to build initially
- $O(\log n)$ to remove an item
- Each iteration therefore takes $O(\log n)$
- All n iterations take $O(n \log n)$

Question for you

- Is it correct to say that the running time of the algorithm is $\Theta(n \log n)$?

Question for you

- Is it correct to say that the running time of the algorithm is $\Theta(n \log n)$?

No. Consider the case where all intervals end at same time. The body of loop takes constant time because removal from heap never involves any exchanges.

Question for you

- What would the running time of the algorithm be if we implement the priority queue as an unsorted list?

Question for you

- What would the running time of the algorithm be if we implement the priority queue as an unsorted list?

Running time would be $O(n^2)$ since it would take $O(n)$ time to find highest priority element

Question for you

- Consider best-case and worst-case running times.

Question for you

- Consider best-case and worst-case running times.

heap: Best-case $O(n)$, worst-case $O(n \log n)$

unsorted list: Best-case $O(n)$, worst-case $O(n^2)$

Greedy Algorithms: The Shortest Path Problem

Shortest Path Problem

Problem: Given a weighted graph $G = (V, E)$, an edge weight function w and some $s \in V$ the goal is to find the length of the shortest path to each vertex in V

Definitions:

- A **path** p is a sequence of vertices $p = (v_1, \ldots, v_k)$ where

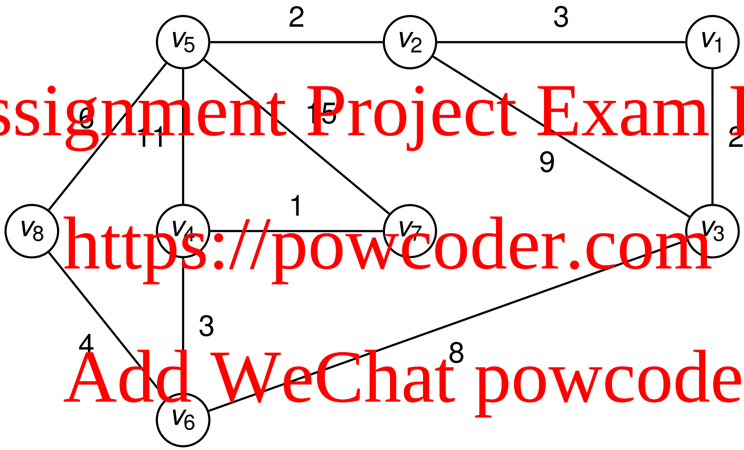
$$\{v_i, v_{i+1}\} \in E \qquad \text{for all } i \text{ such that } 1 \leq i < k$$

- The **weight of a path** is the sum of the weights of the edges on the path

- The **distance** of a vertex v from s is the weight of the shortest (least weighted) path from s to v

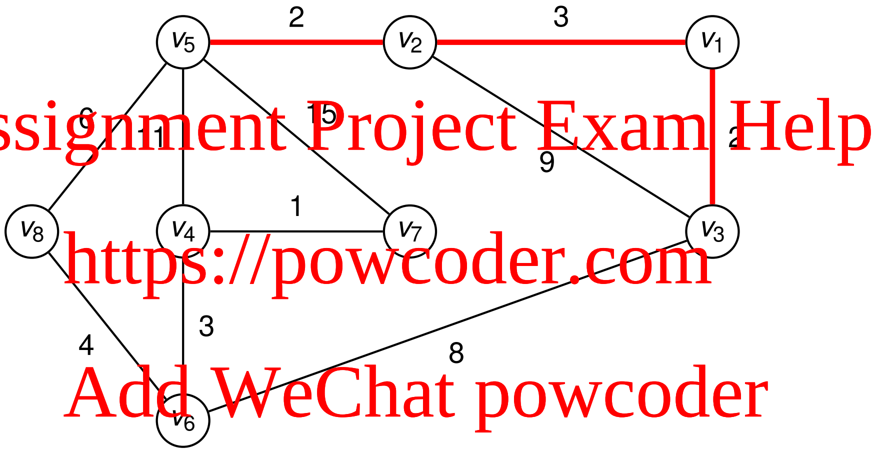
Note that we will assume that all edge weights are positive

Example Graph



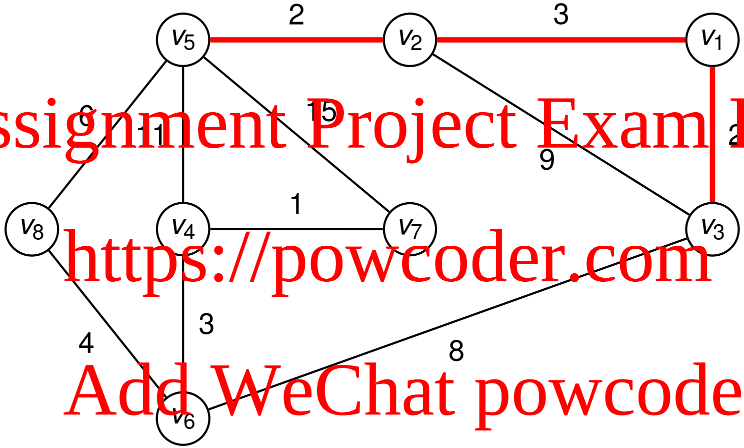
Consider the path (v_5, v_2, v_1, v_3)

Example Graph



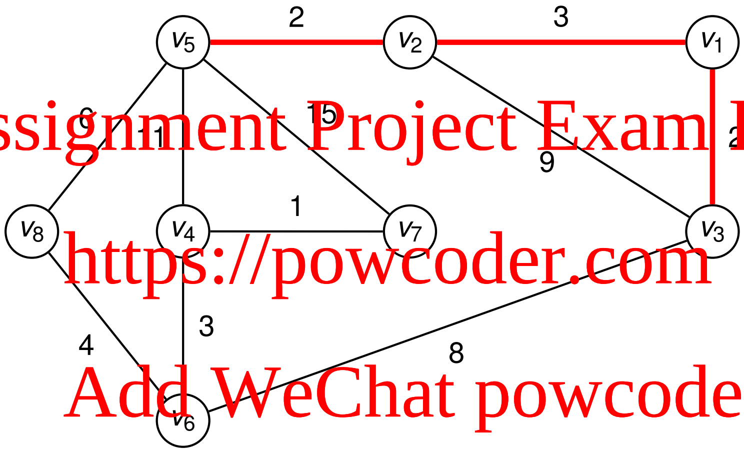
Consider the path (v_5, v_2, v_1, v_3)

Example Graph



The weight of (v_5, v_2, v_1, v_3) is $2 + 3 + 2 = 7$

Example Graph



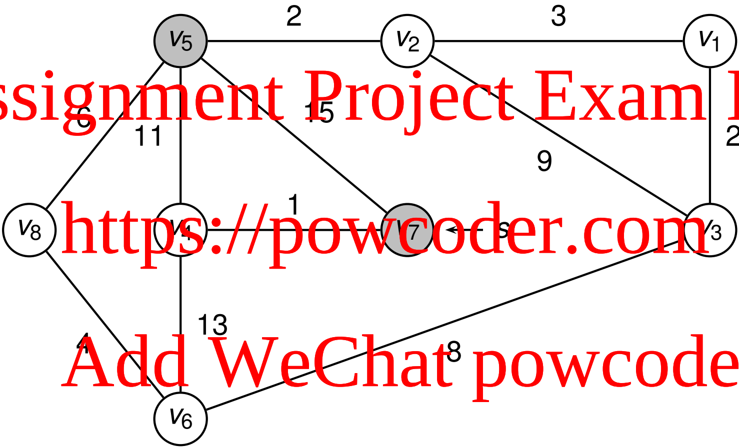
(v_5, v_2, v_1, v_3) is the shortest path from v_5 to v_3

A Greedy Algorithm

The underlying algorithm

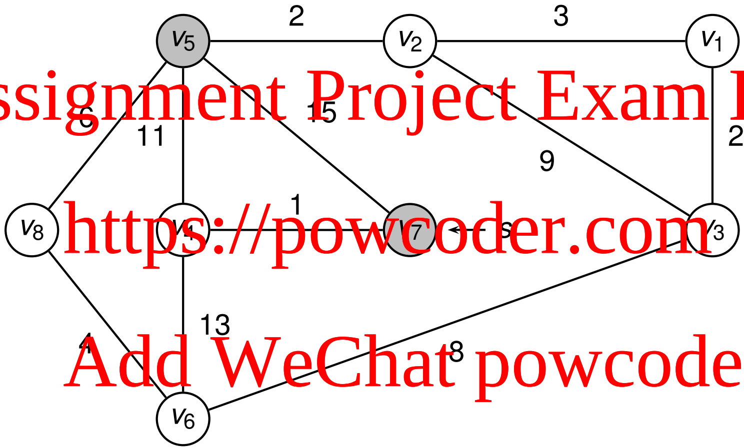
- Vertices added to a set A once their distance from s established
- For vertices v not yet in A we keep track of $\delta(v)$: the shortest distance from s to v routed only through vertices that are already in A

Example



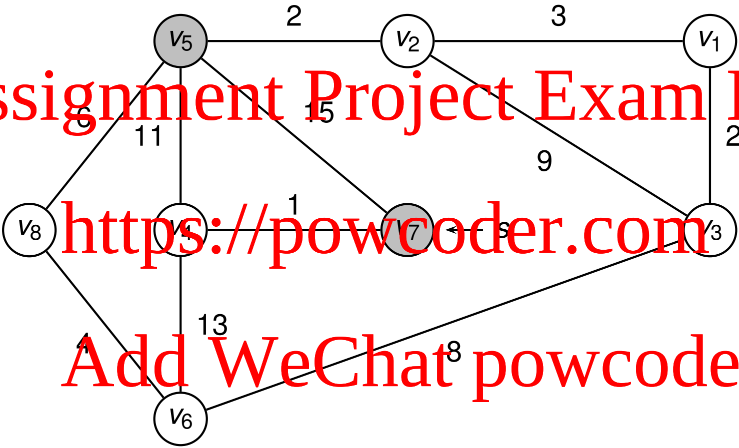
Suppose $s = v_7$ and $A = \{v_5, v_7\}$

Example



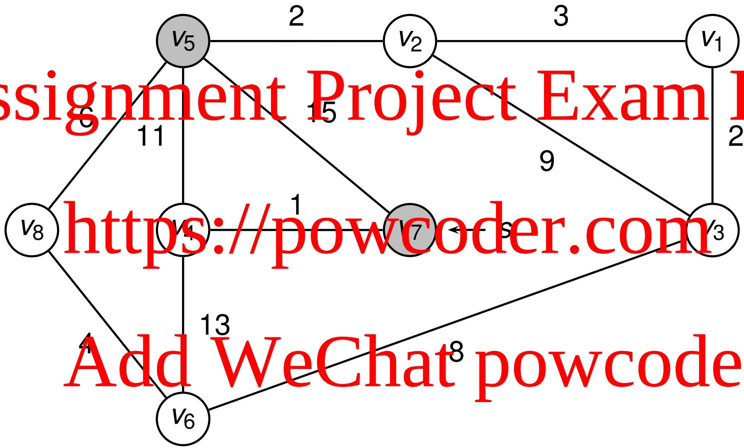
$\delta(v_7) = 0$

Example



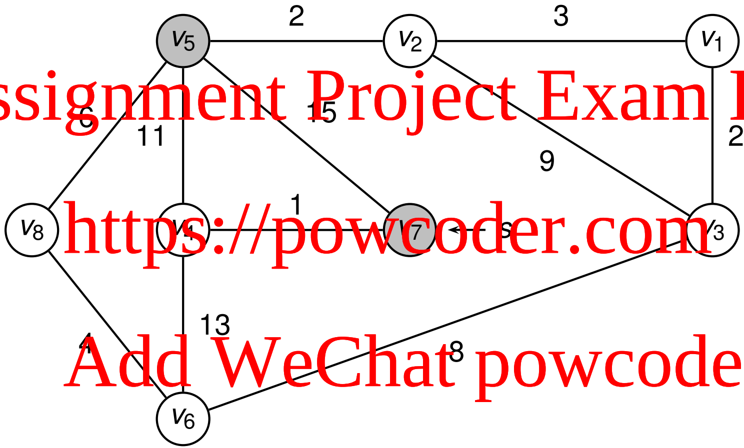
$\delta(v_5) = 15$

Example



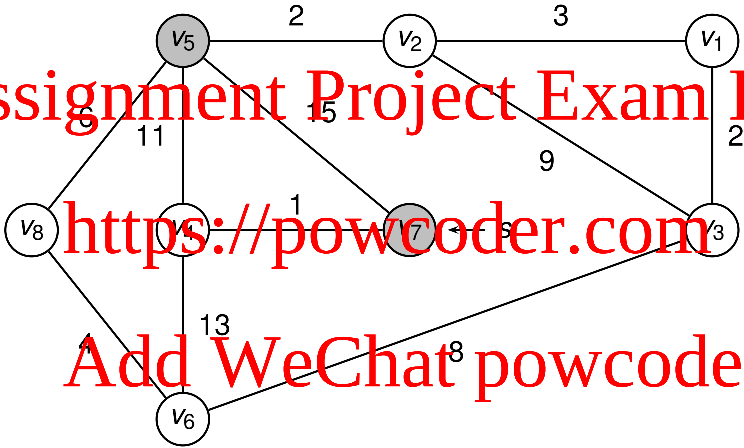
$\delta(v_4) = 1$

Example



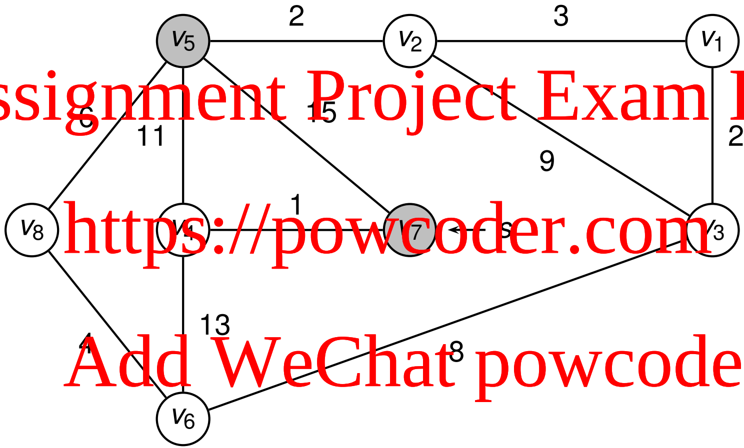
$\delta(v_2) = 17$

Example



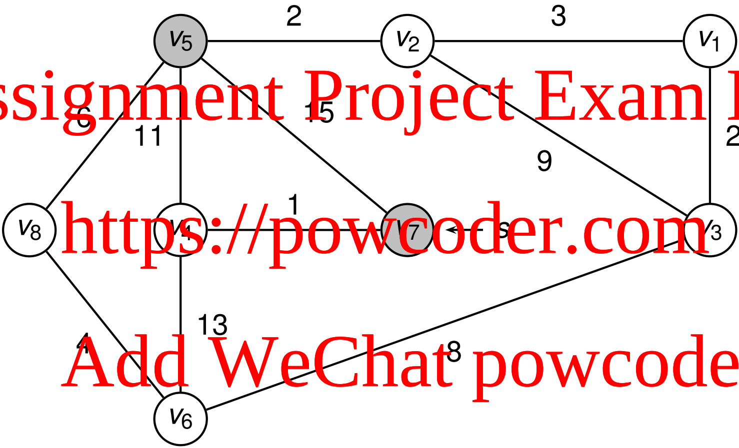
$\delta(v_8) = 21$

Example



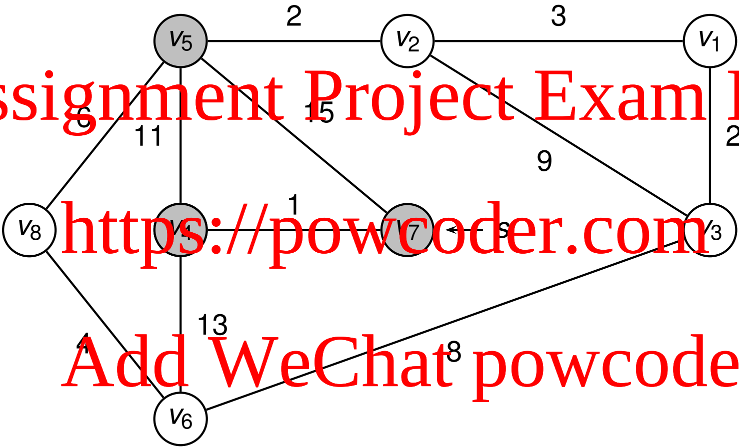
For all other vertices v we have $\delta(v) = \infty$

Example



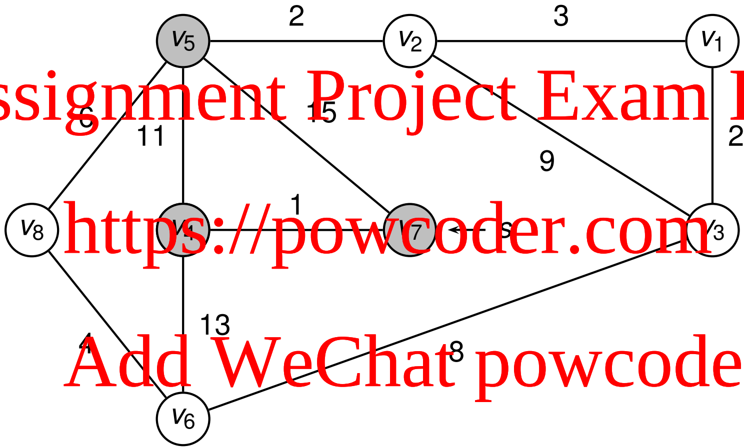
Suppose we add v_4 to A

Example



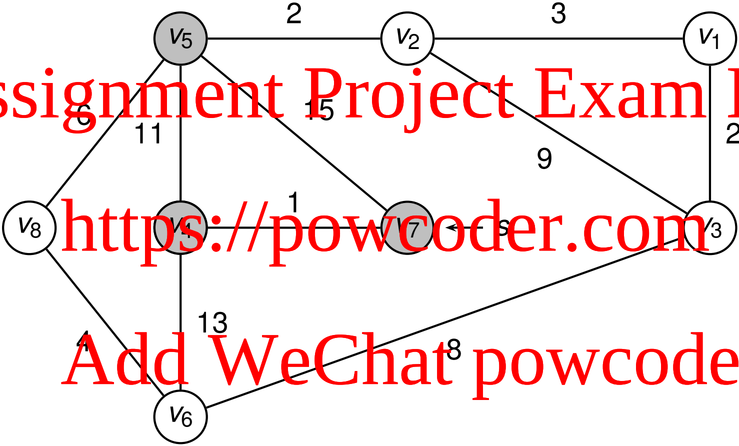
Potentially changes values of δ for vertices adjacent to v_4

Example



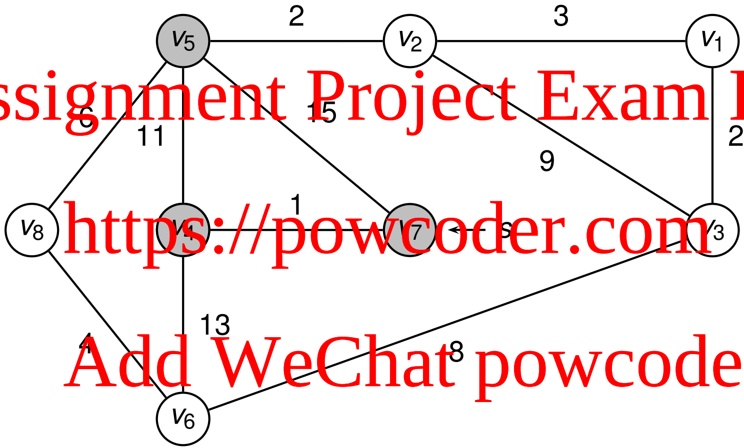
$\delta(v_5)$ reduces to 12

Example



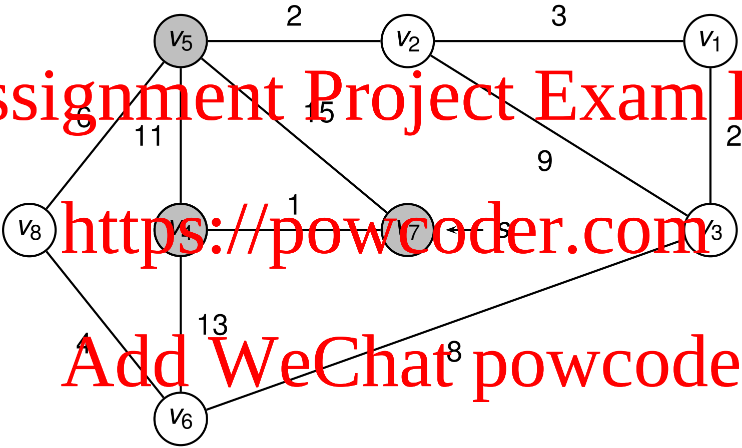
$\delta(v_6) = 14$

Example



$\delta(v_7)$ isn't improved

Example

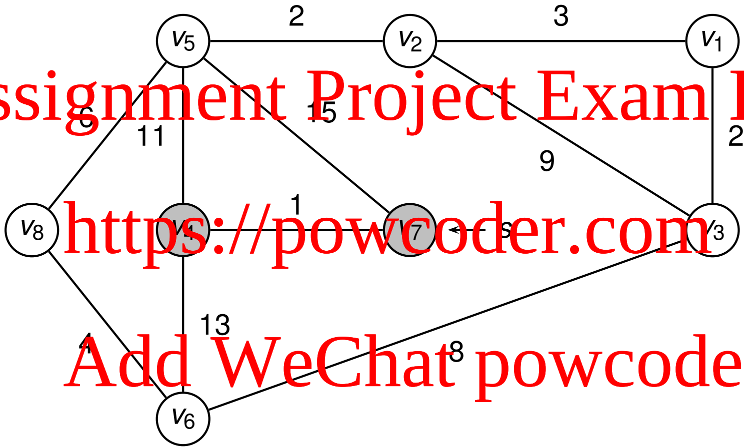


But note that the value $\delta(v_2)$ can be improved from 17 to 14

Building up A

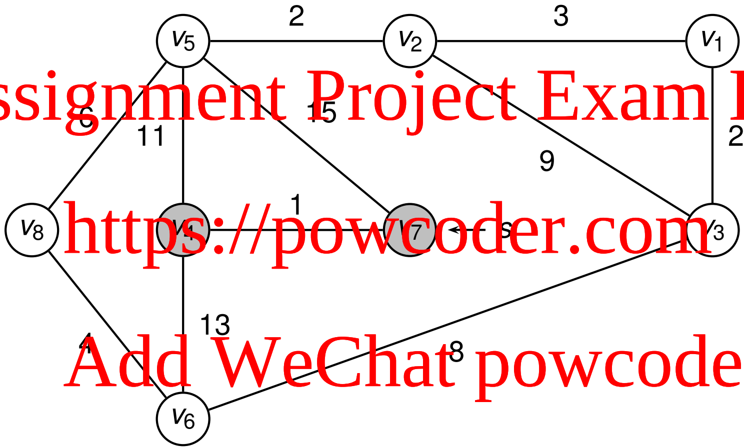
- The order we add vertices to A is crucial
- We must add vertices in order of increasing distance from s
- We will only need to consider δ values for vertices not yet in A

Example Reconsidered



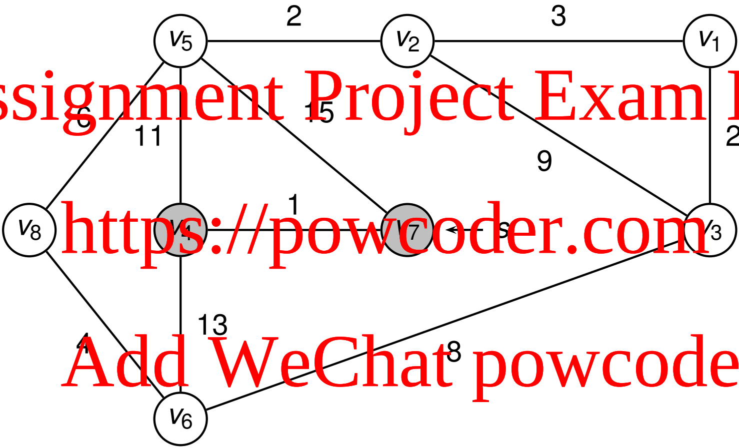
What happens if we add v_4 to A before v_5

Example Reconsidered



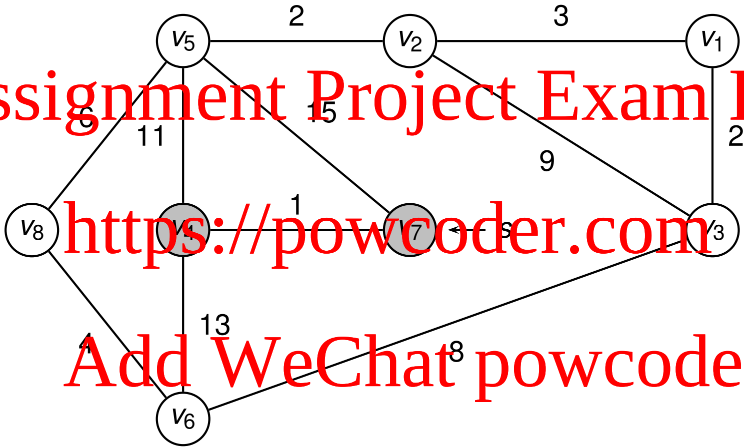
$\delta(v_7) = 0$

Example Reconsidered



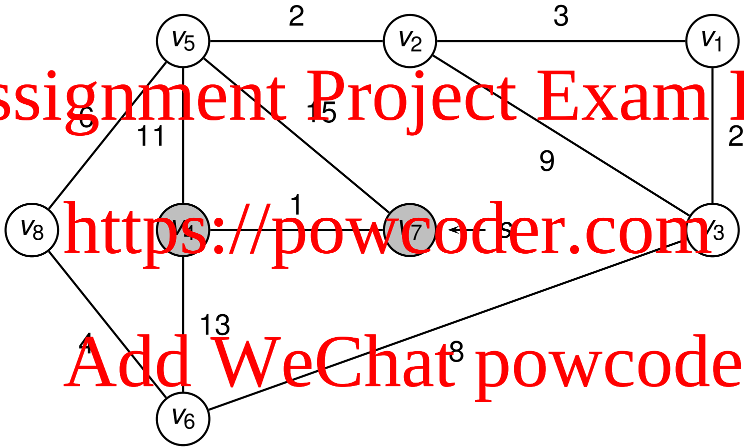
$\delta(v_4) = 1$

Example Reconsidered



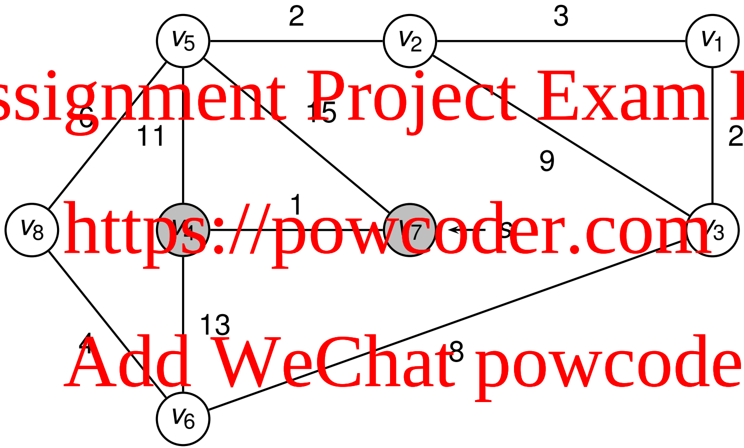
$\delta(v_5) = 12$

Example Reconsidered



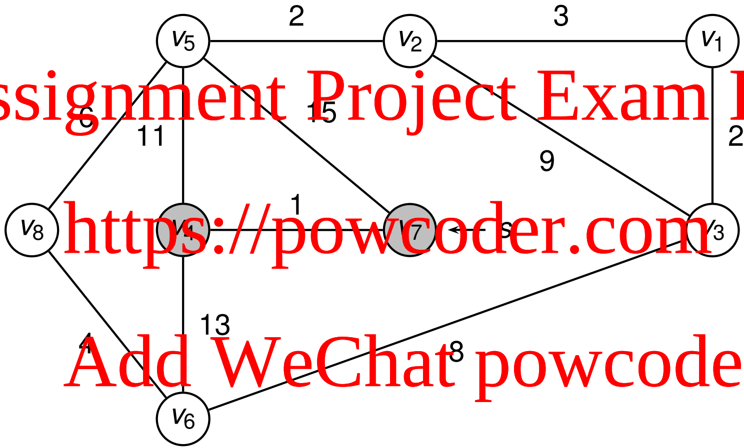
$\delta(v_6) = 14$

Example Reconsidered



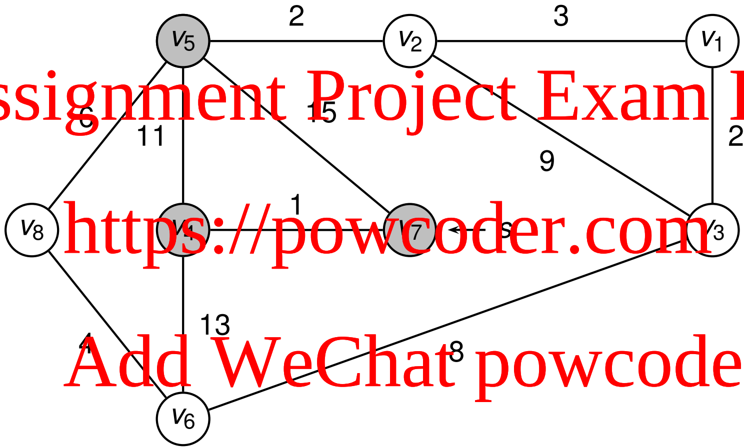
For all other vertices v we have $\delta(v) = \infty$

Example Reconsidered



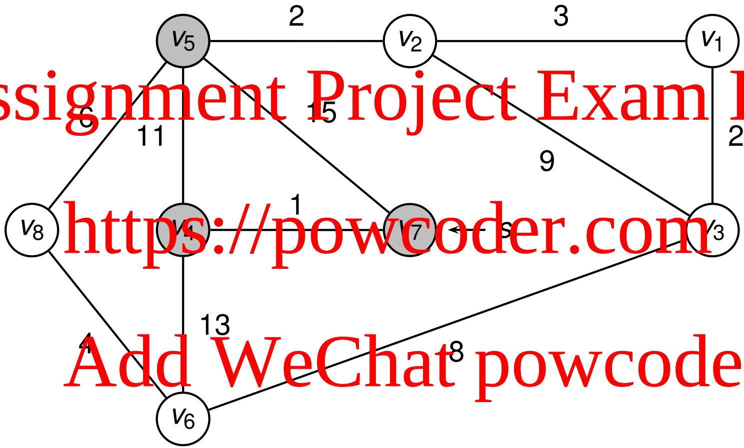
Suppose we now add v_5 to A

Example Reconsidered



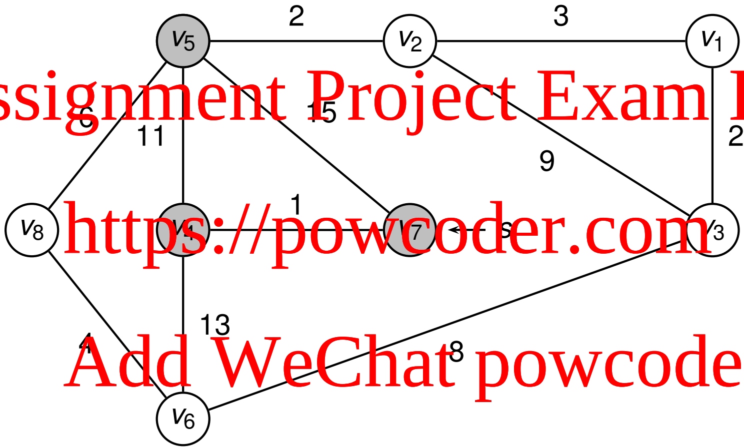
Potentially changes values of δ for vertices adjacent to v_5

Example Reconsidered



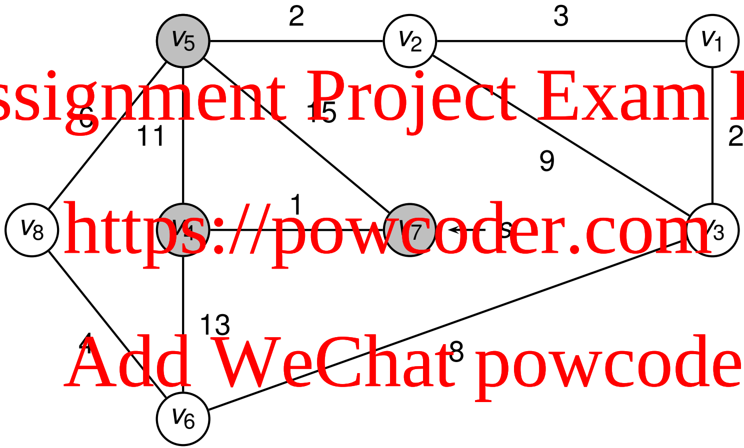
$\delta(v_5)$ can't be improved

Example Reconsidered



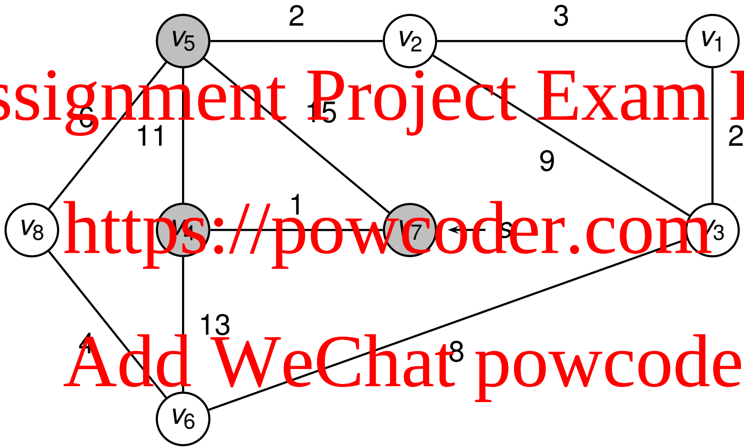
$\delta(v_7)$ can't be improved

Example Reconsidered



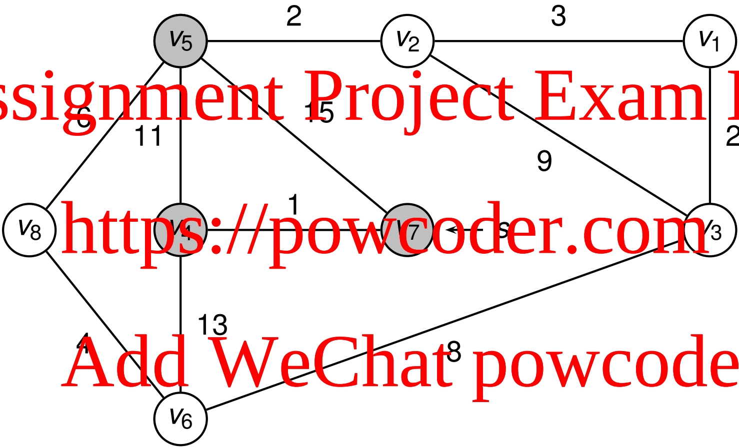
$\delta(v_2) = 14$

Example Reconsidered



$\delta(v_8) = 18$

Example Reconsidered



This time it was sufficient *just* to consider vertices adjacent to v_5

Prioritising Vertices

Use a priority queue prioritised by δ values

- We need to select vertices in order of distance from s
- We can use δ values for this
- Some δ values will need updating
- δ value for vertices or vertex nearest to s can't be improved
- These can safely be added to A
- We'll prove this later!

Dijkstra’s Algorithm

Dijkstra(G,w,s):

let A be the empty set

let $\delta(s) = 0$

let $\delta(v) = \infty$ *for* $v \in V - \{s\}$

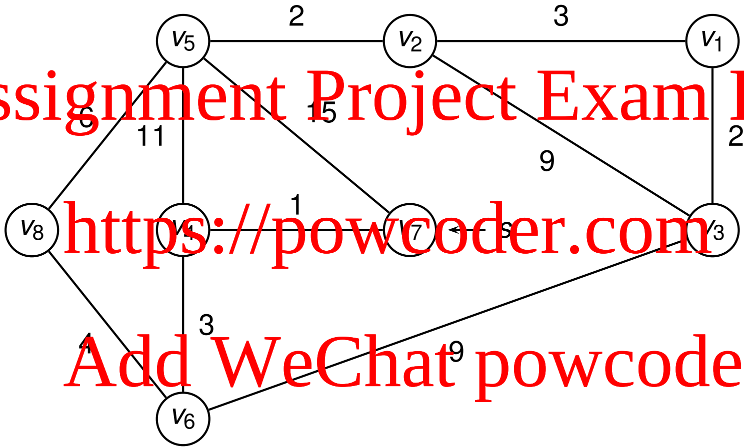
let Q be a priority queue containing elements of V

while Q is not empty

- *remove v from front of priority queue Q*
- *add v to A*
- *for each* $\langle v, u \rangle \in E$ *where* $u \notin A$

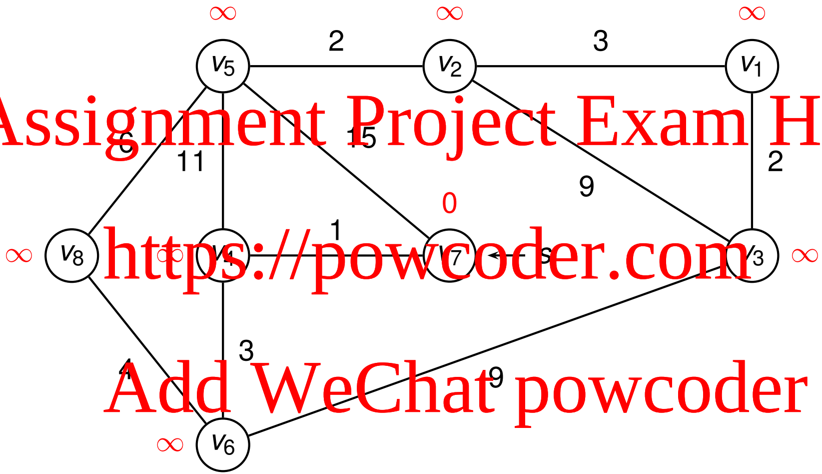
  - *if* $\delta(u) > \delta(v) + w(v, u)$ *then*
    - *let* $\delta(u) = \delta(v) + w(v, u)$

Illustration of Dijkstra's Algorithm



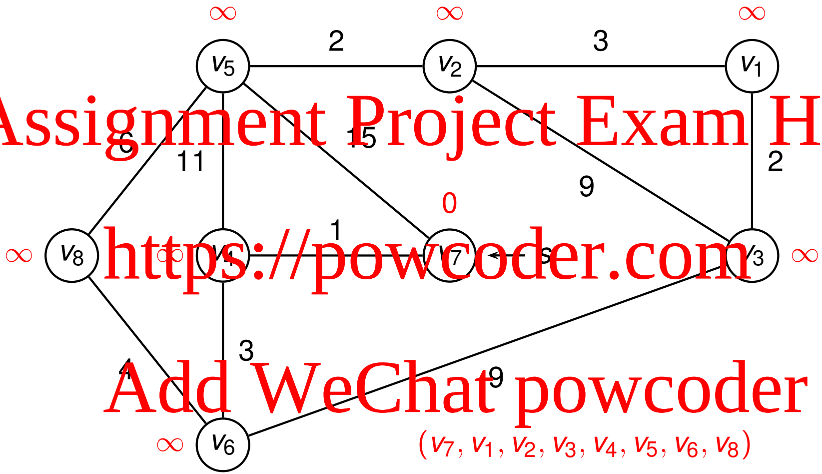
Initialise δ values

Illustration of Dijkstra's Algorithm



Initialise Q

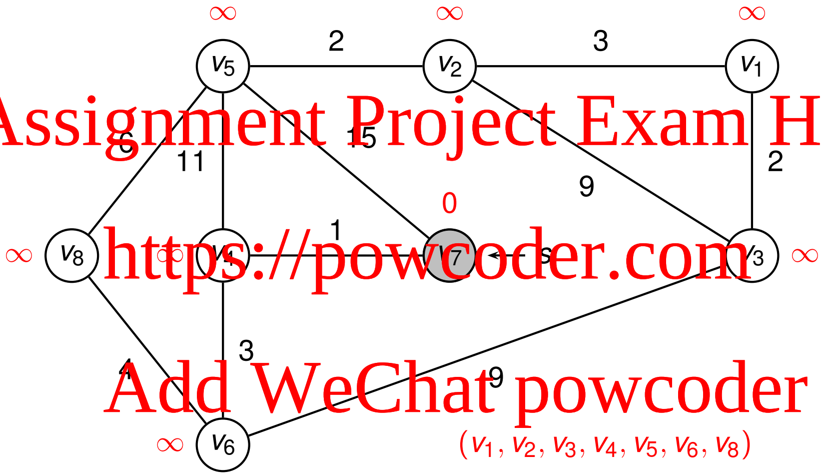
Illustration of Dijkstra's Algorithm



$(v_7, v_1, v_2, v_3, v_4, v_5, v_6, v_8)$

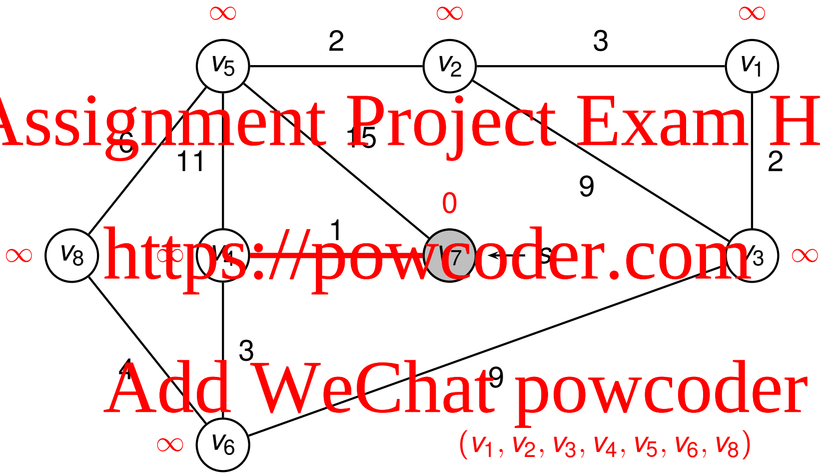
Remove v_7 from Q and add to A

Illustration of Dijkstra's Algorithm



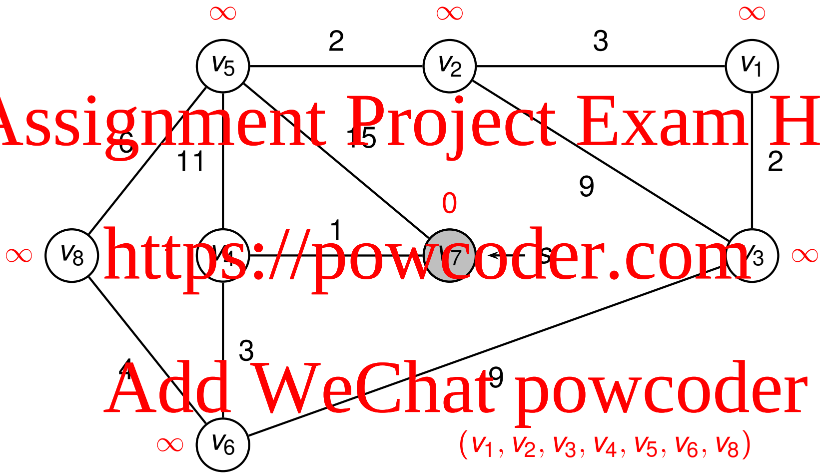
Consider edge $\{v_7, v_4\}$

Illustration of Dijkstra’s Algorithm



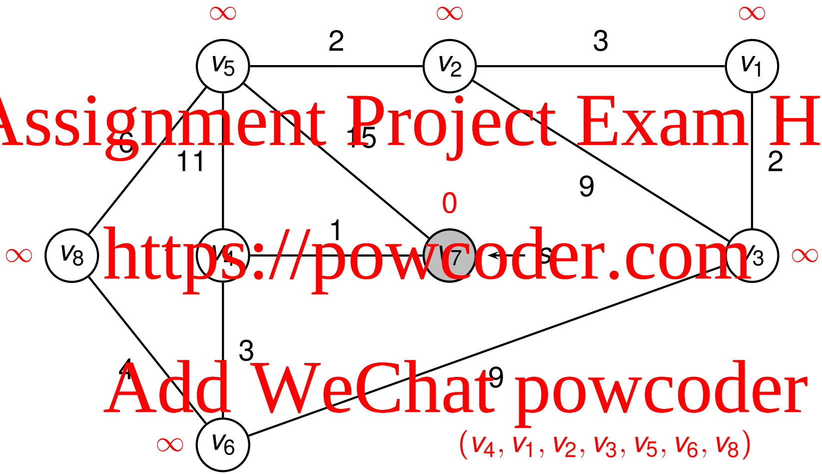
$\delta(v_4) > \delta(v_7) + w(v_7, v_4)$ so update $\delta(v_4)$

Illustration of Dijkstra's Algorithm



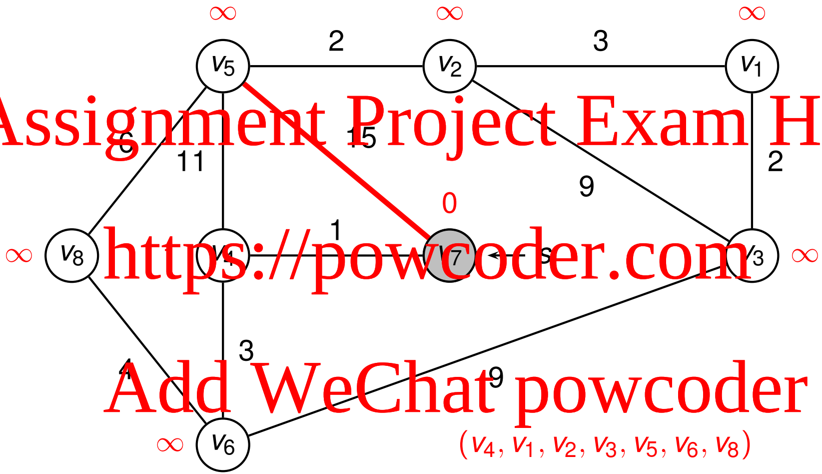
Reposition v_4 in Q

Illustration of Dijkstra’s Algorithm



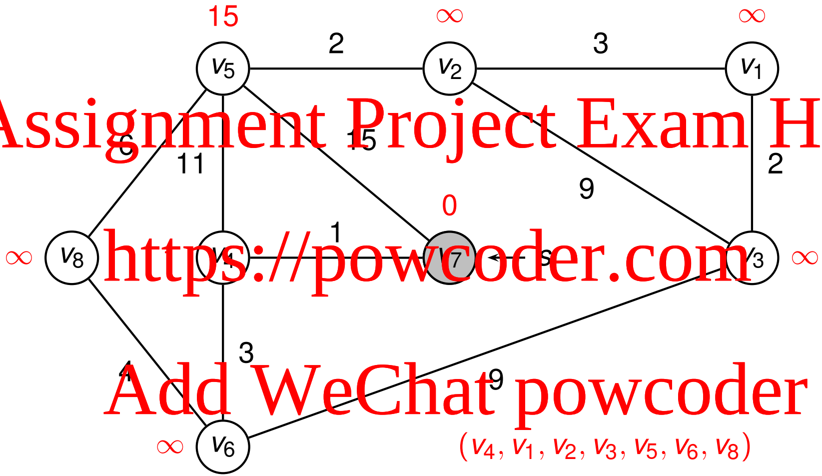
Consider edge $\{v_7, v_5\}$

Illustration of Dijkstra's Algorithm



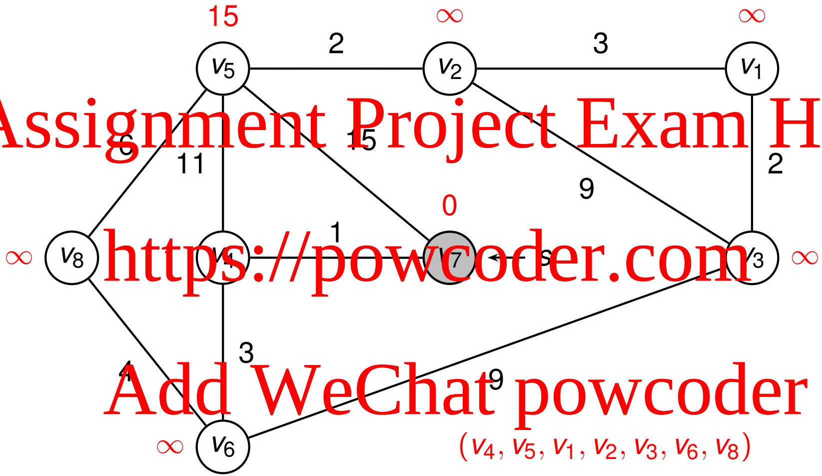
$\delta(v_5) > \delta(v_7) + w(v_7, v_5)$ so update $\delta(v_5)$

Illustration of Dijkstra's Algorithm



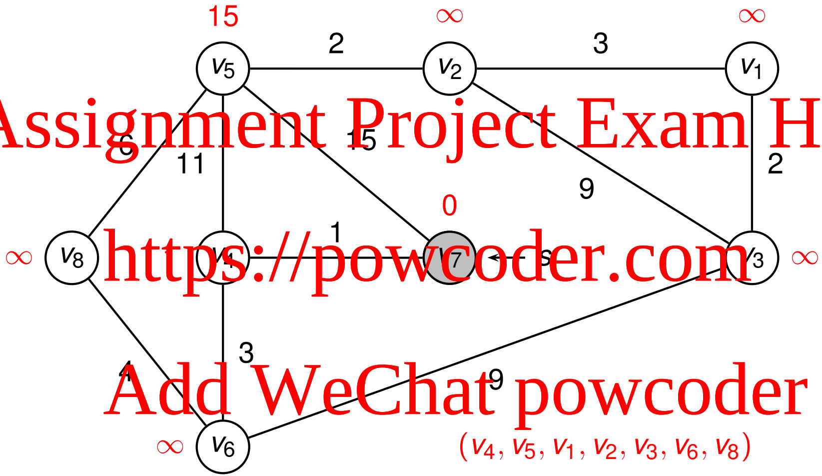
Reposition v_5 in Q

Illustration of Dijkstra's Algorithm



All vertices adjacent to v_7 now considered

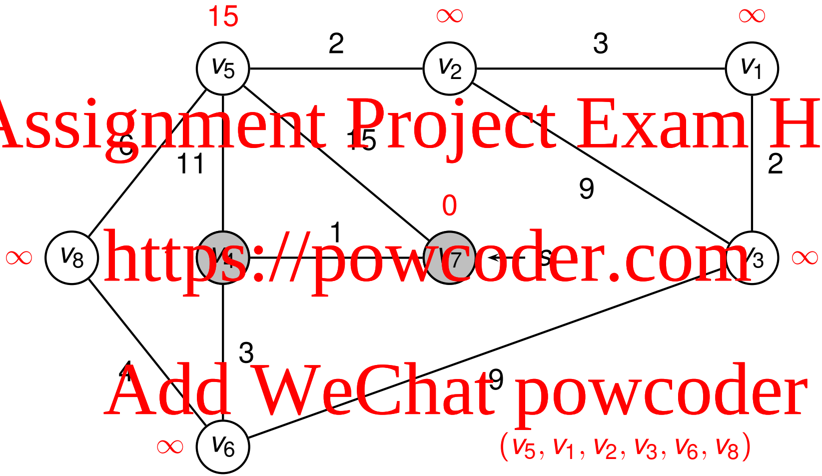
Illustration of Dijkstra's Algorithm



$(v_4, v_5, v_1, v_2, v_3, v_6, v_8)$

Remove v_4 from Q and add to A

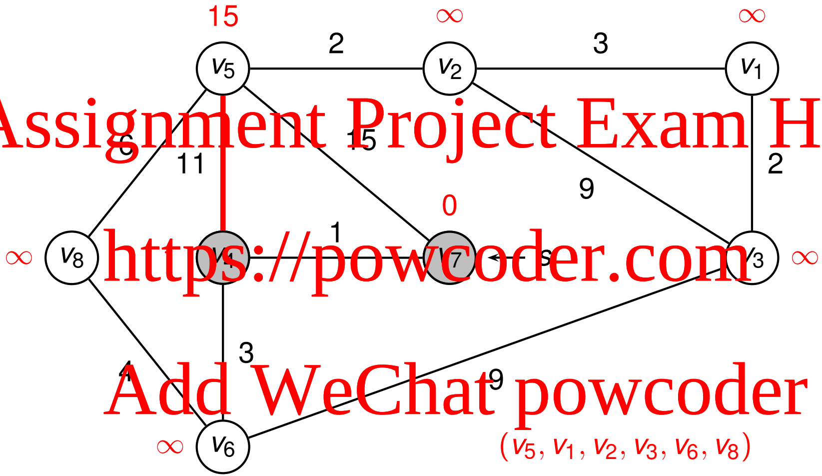
Illustration of Dijkstra's Algorithm



$(v_5, v_1, v_2, v_3, v_6, v_8)$

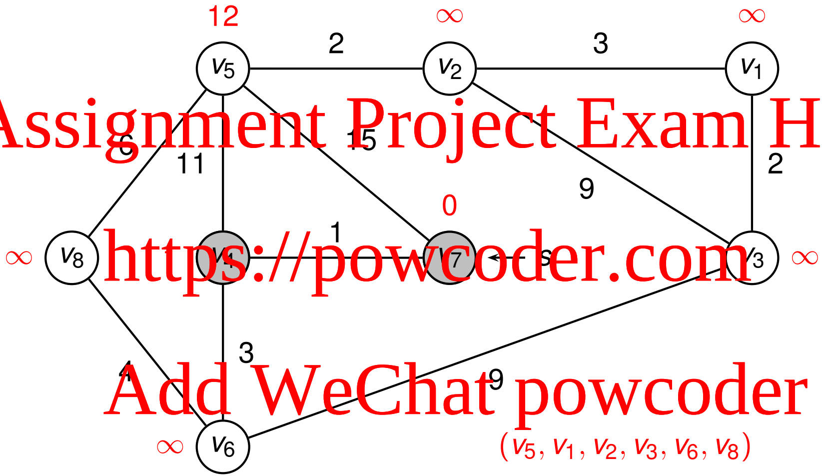
Consider edge $\{v_4, v_5\}$

Illustration of Dijkstra's Algorithm



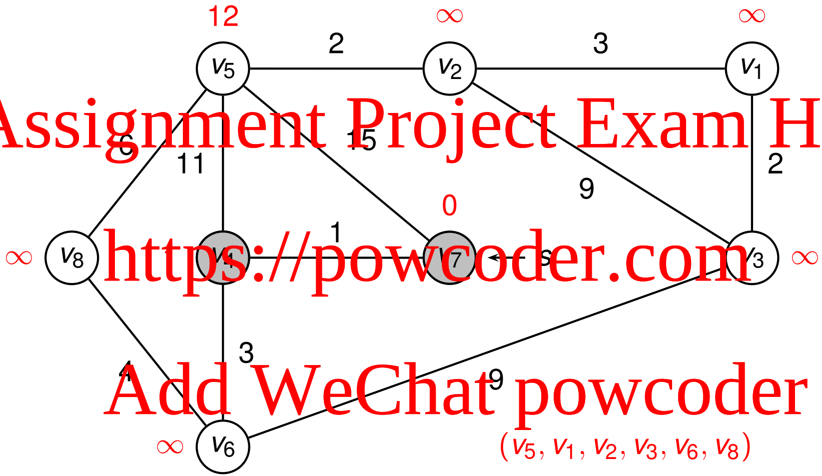
$\delta(v_5) > \delta(v_4) + w(v_4, v_5)$ so update $\delta(v_5)$

Illustration of Dijkstra’s Algorithm



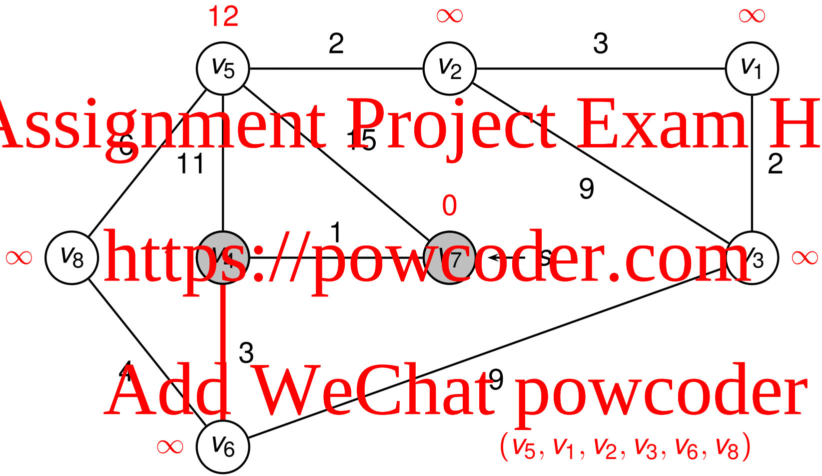
No need to reposition v_5 in Q

Illustration of Dijkstra's Algorithm



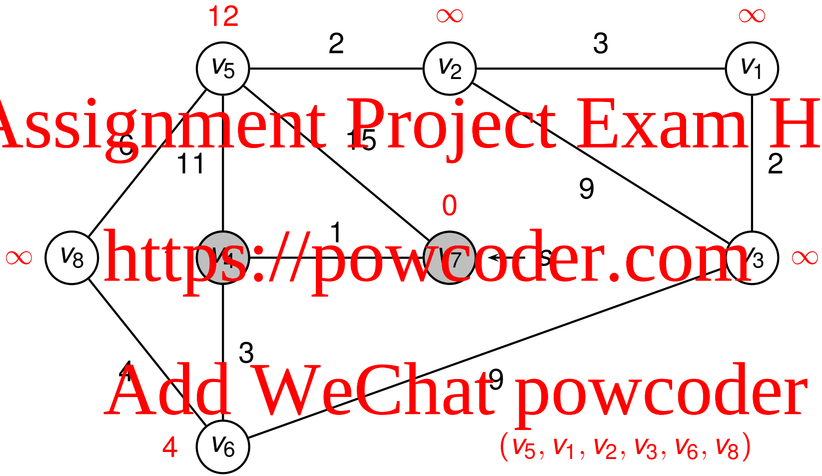
Consider edge $\{v_4, v_6\}$

Illustration of Dijkstra's Algorithm



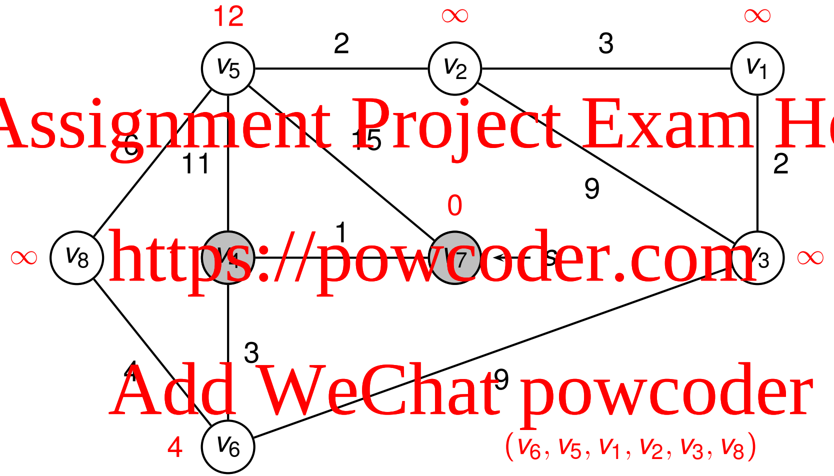
$\delta(v_6) > \delta(v_4) + w(v_4, v_6)$ so update $\delta(v_6)$

Illustration of Dijkstra's Algorithm



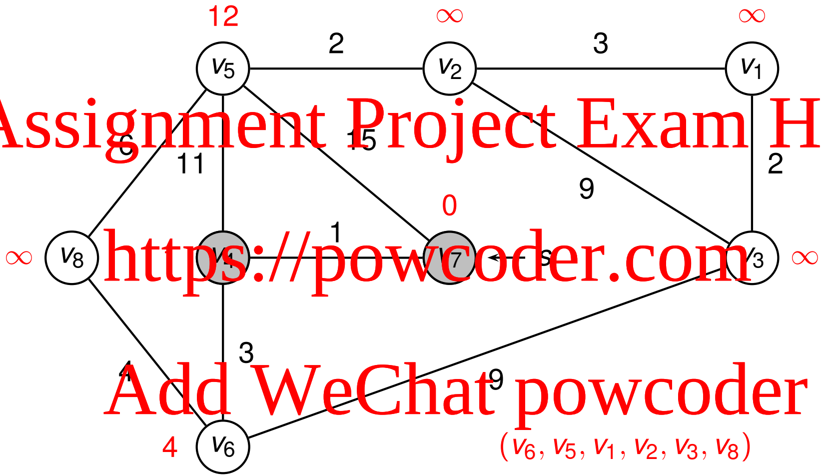
Reposition v_6 in Q

Illustration of Dijkstra's Algorithm



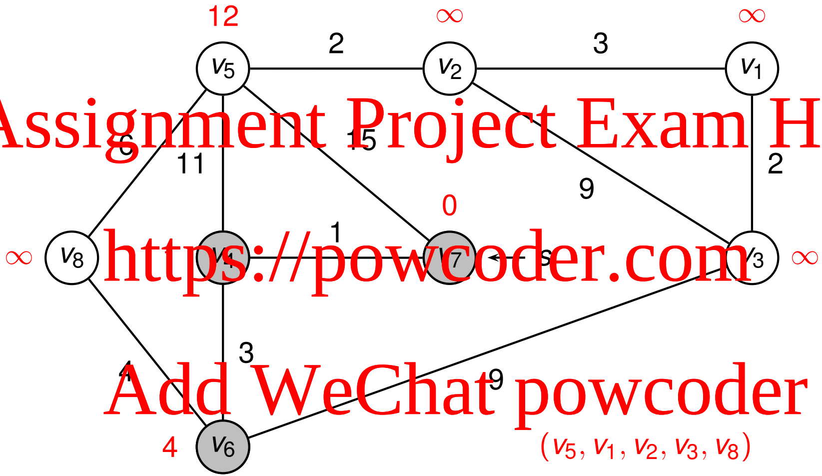
All vertices adjacent to v_4 now considered

Illustration of Dijkstra’s Algorithm



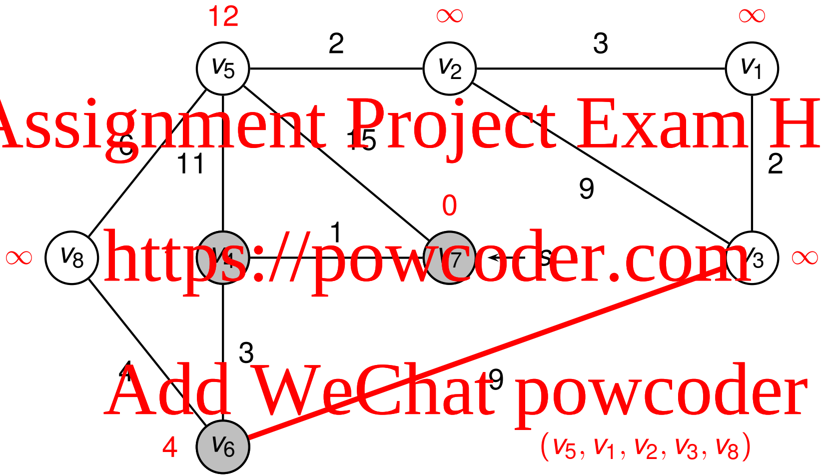
Remove v_6 from Q and add to A

Illustration of Dijkstra's Algorithm



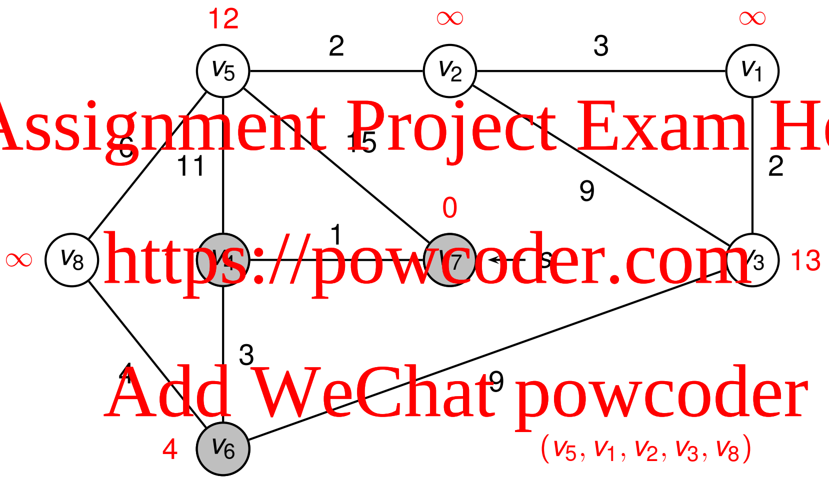
Consider edge $\{v_6, v_3\}$

Illustration of Dijkstra's Algorithm



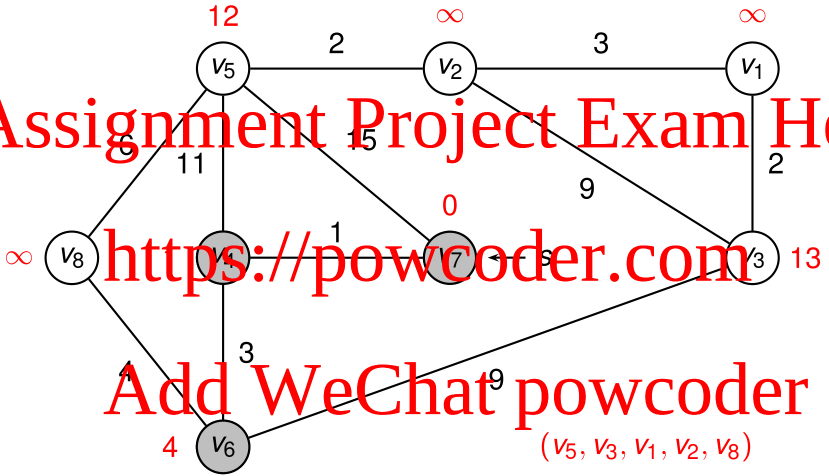
$\delta(v_3) > \delta(v_6) + w(v_6, v_3)$ so update $\delta(v_3)$

Illustration of Dijkstra's Algorithm



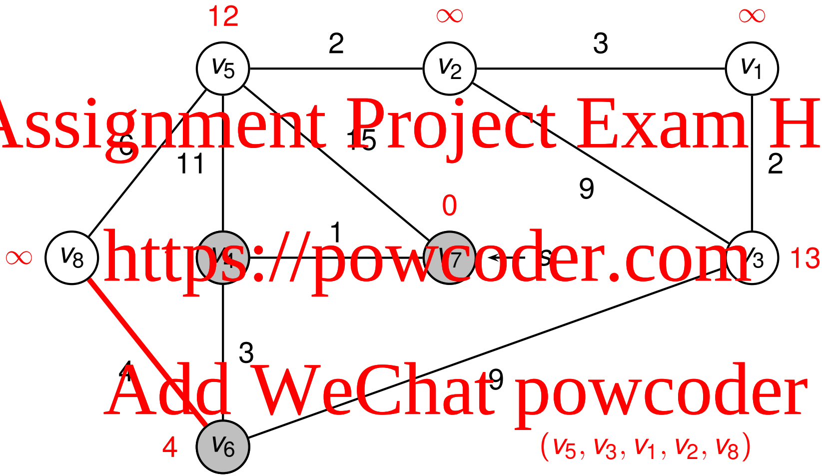
Reposition v_3 in Q

Illustration of Dijkstra's Algorithm



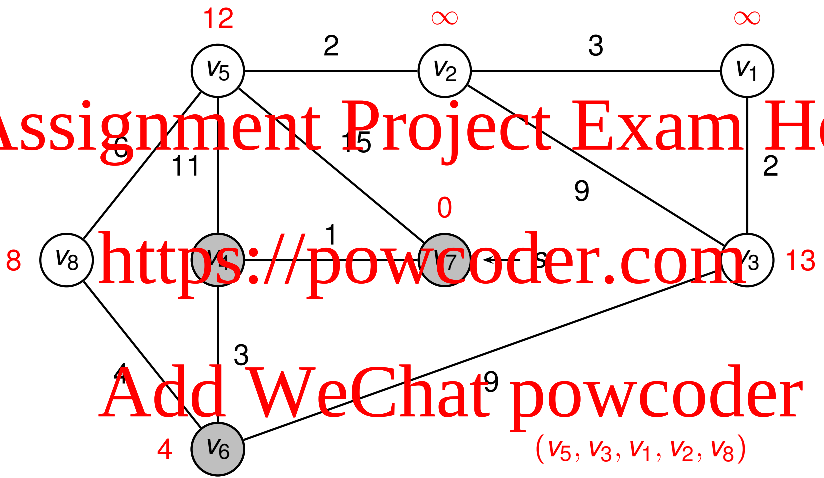
Consider edge $\{v_6, v_8\}$

Illustration of Dijkstra's Algorithm



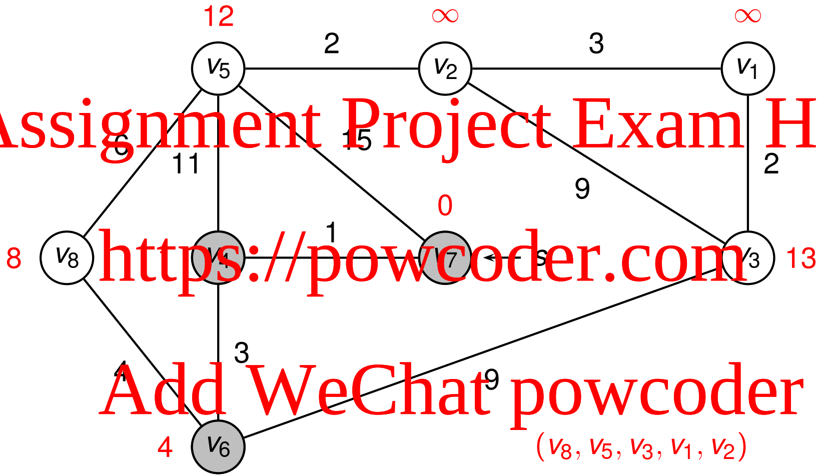
$\delta(v_8) > \delta(v_6) + w(v_6, v_8)$ so update $\delta(v_8)$

Illustration of Dijkstra's Algorithm



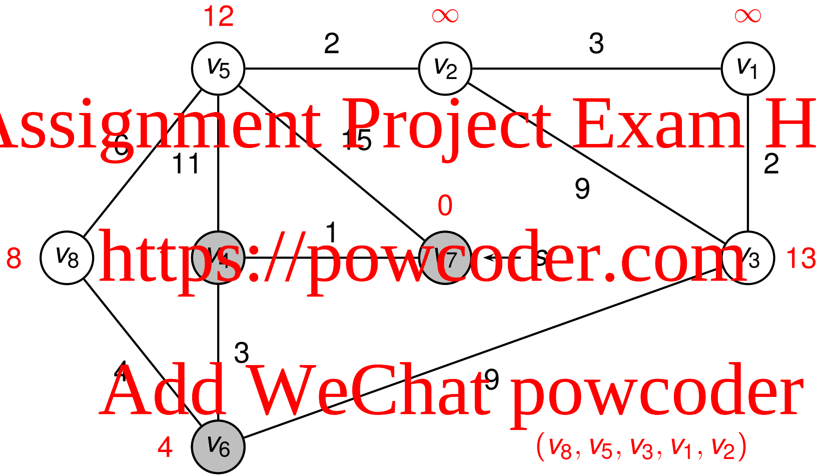
Reposition v_8 in Q

Illustration of Dijkstra's Algorithm



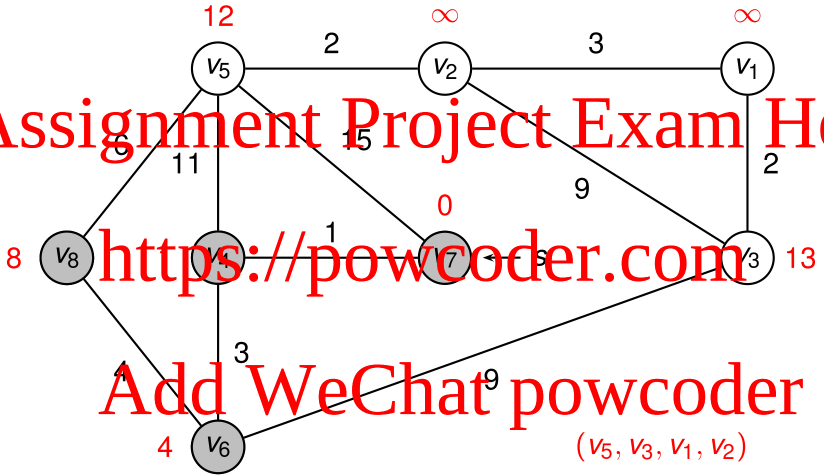
All vertices adjacent to v_6 now considered

Illustration of Dijkstra's Algorithm



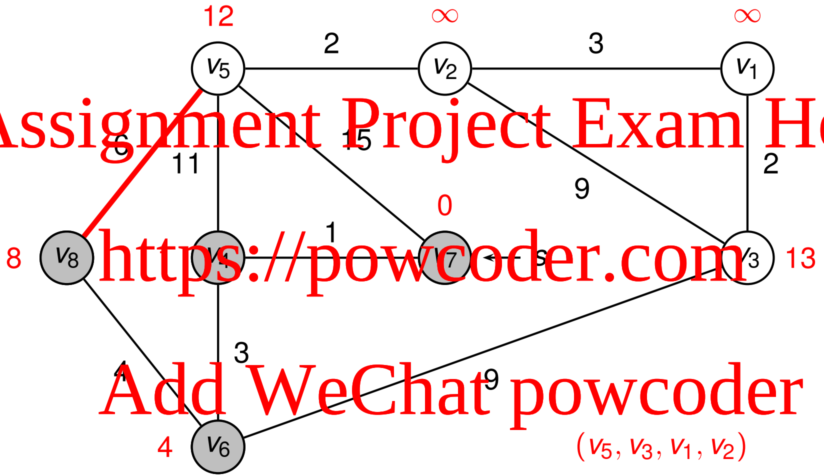
Remove v_8 from Q and add to A

Illustration of Dijkstra's Algorithm



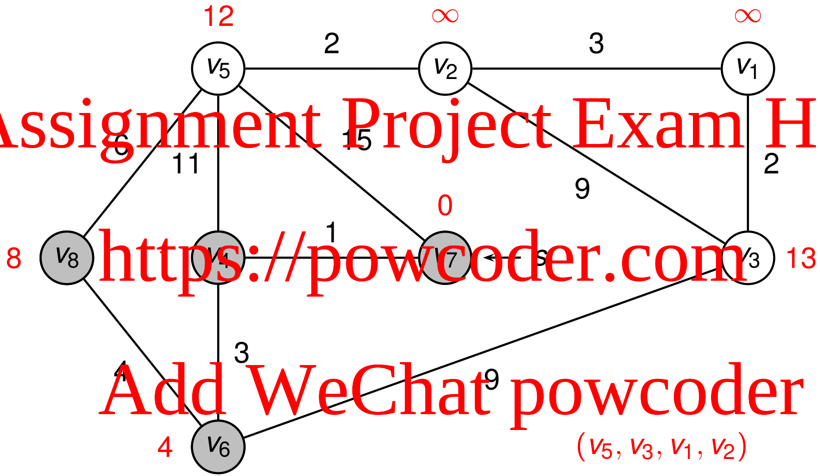
Consider edge $\{v_8, v_5\}$

Illustration of Dijkstra's Algorithm



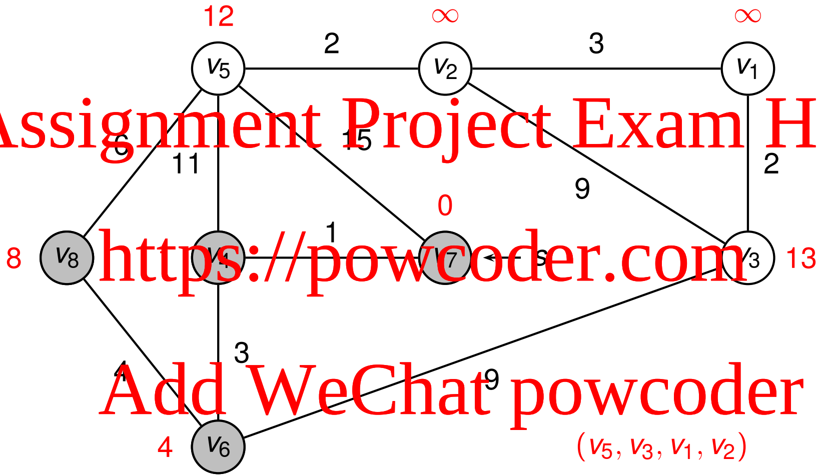
$\delta(v_5) < \delta(v_8) + w(v_8, v_5)$ so don't update $\delta(v_5)$

Illustration of Dijkstra’s Algorithm



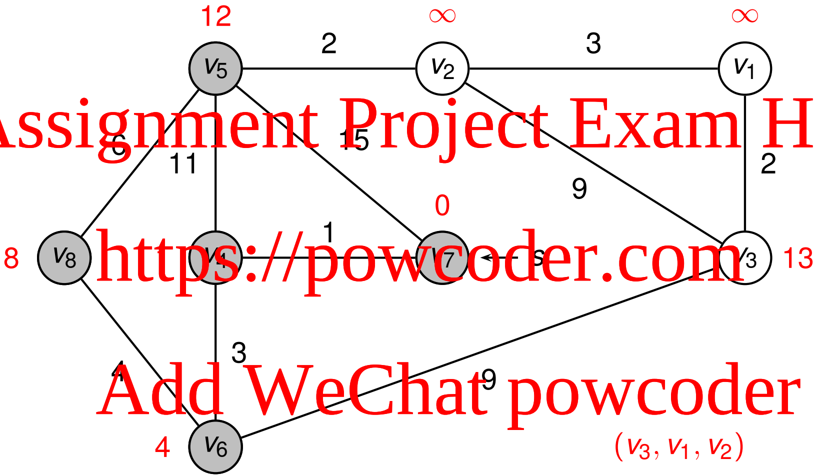
All vertices adjacent to v_8 now considered

Illustration of Dijkstra's Algorithm



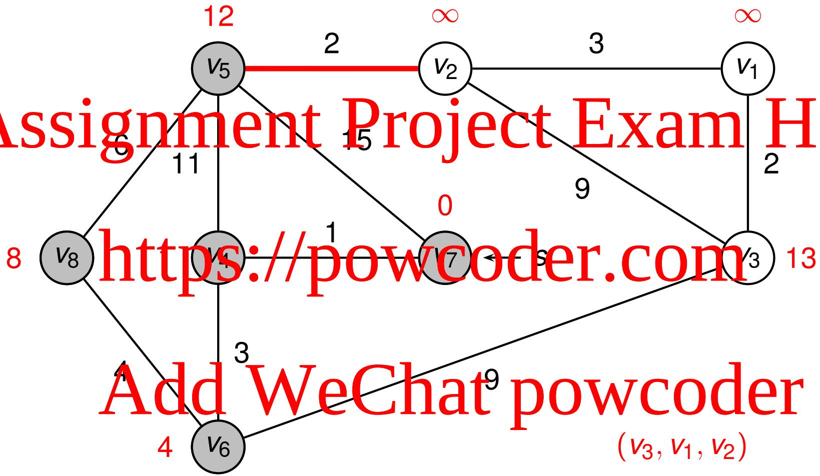
Remove v_5 from Q and add to A

Illustration of Dijkstra's Algorithm



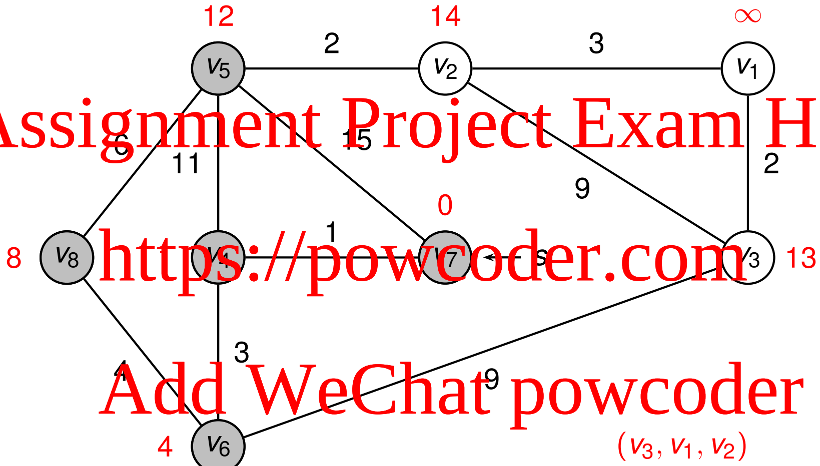
Consider edge $\{v_5, v_2\}$

Illustration of Dijkstra's Algorithm



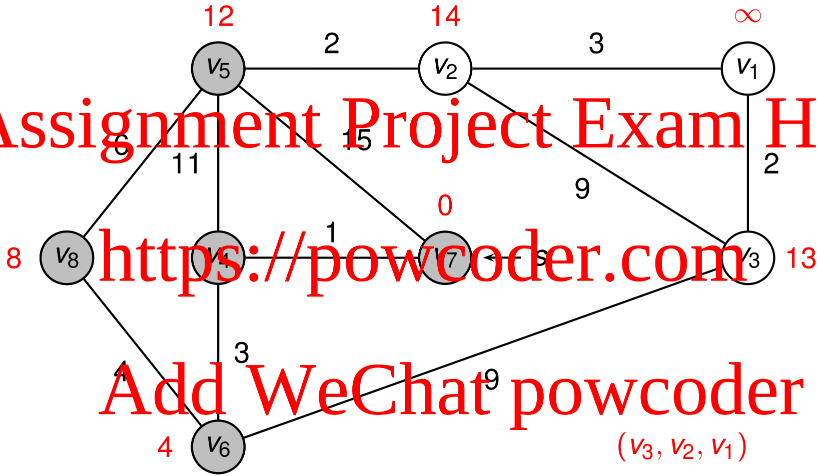
$\delta(v_2) > \delta(v_5) + w(v_5, v_2)$ so update $\delta(v_2)$

Illustration of Dijkstra's Algorithm



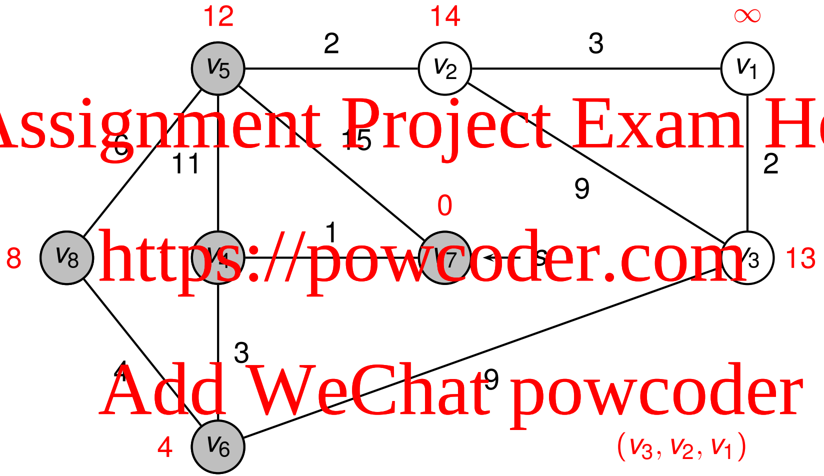
Reposition v_2 in Q

Illustration of Dijkstra's Algorithm



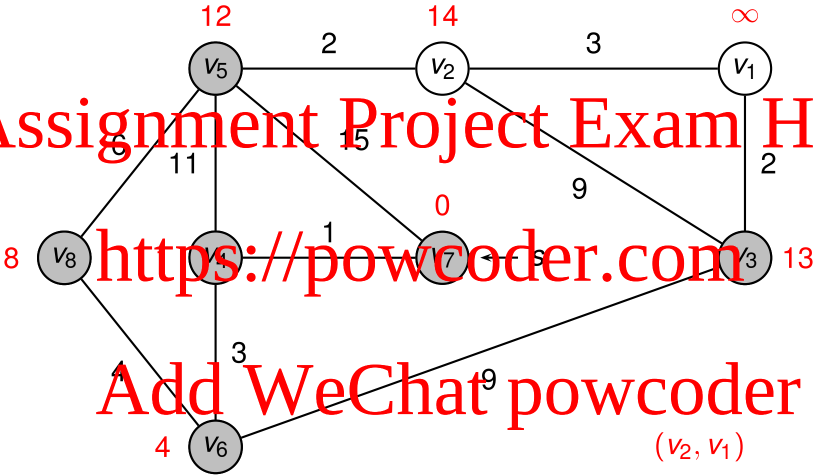
All vertices adjacent to v_5 now considered

Illustration of Dijkstra's Algorithm



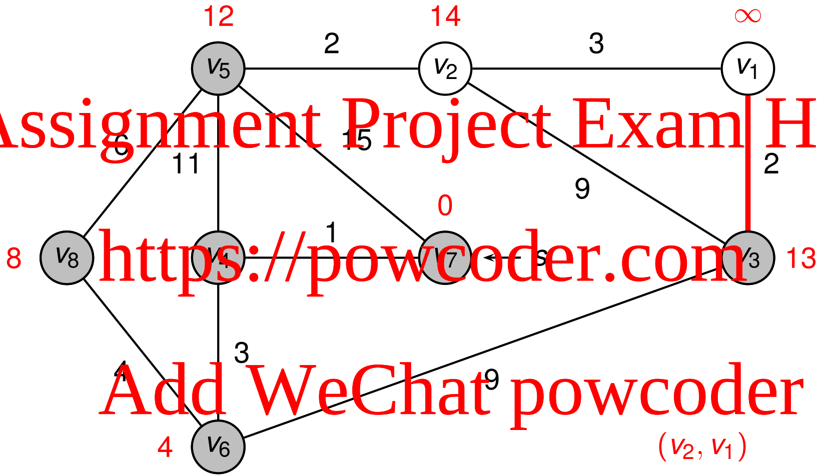
Remove v_3 from Q and add to A

Illustration of Dijkstra's Algorithm



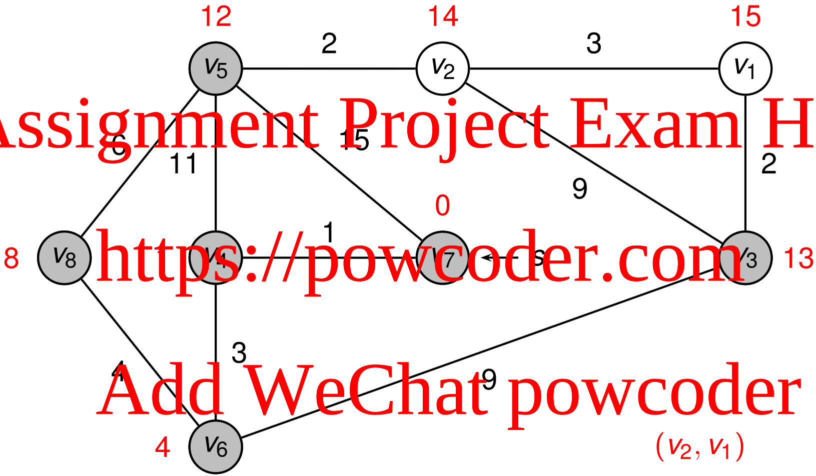
Consider edge $\{v_3, v_1\}$

Illustration of Dijkstra's Algorithm



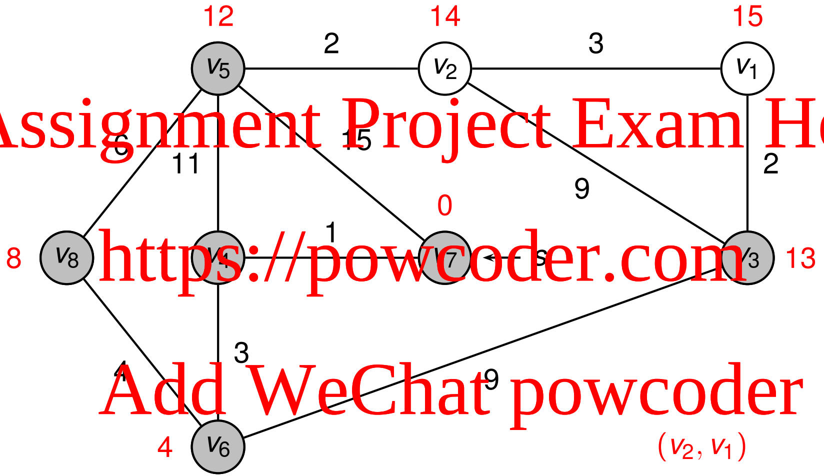
$\delta(v_1) > \delta(v_3) + w(v_3, v_1)$ so update $\delta(v_1)$

Illustration of Dijkstra’s Algorithm



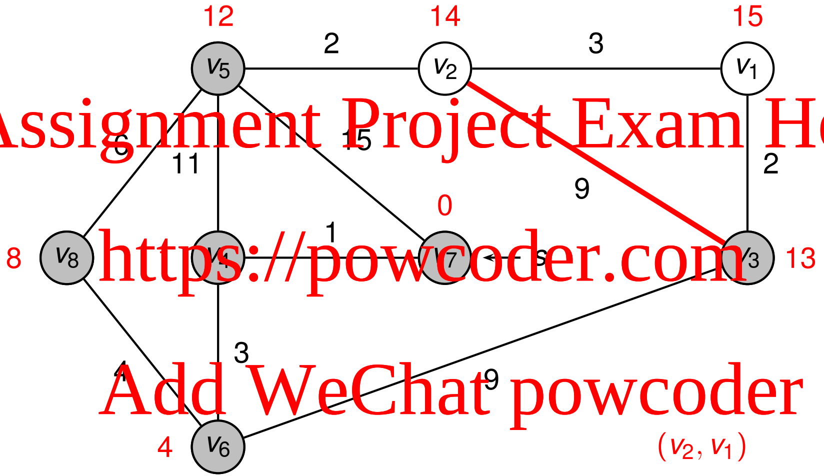
No need to reposition v_1 in Q

Illustration of Dijkstra's Algorithm



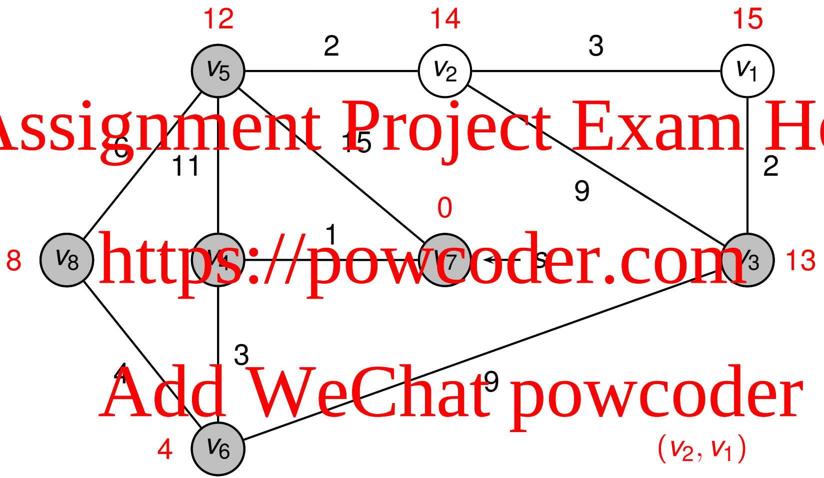
Consider edge $\{v_3, v_2\}$

Illustration of Dijkstra's Algorithm



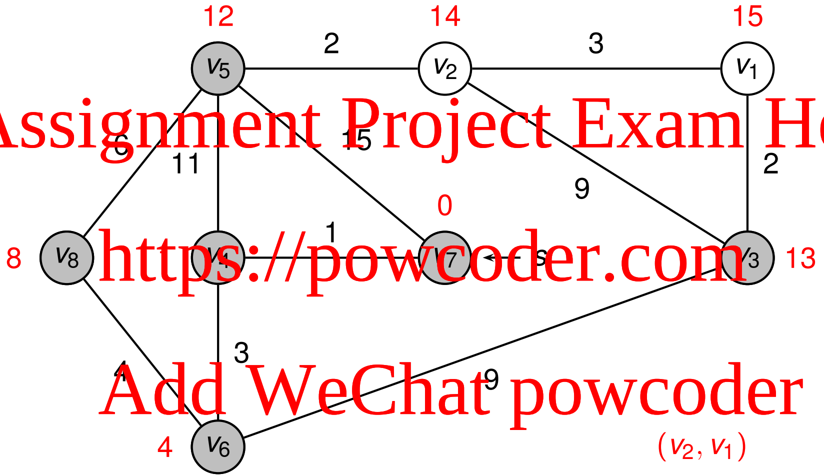
$\delta(v_2) < \delta(v_3) + w(v_3, v_2)$ so don't update $\delta(v_2)$

Illustration of Dijkstra’s Algorithm



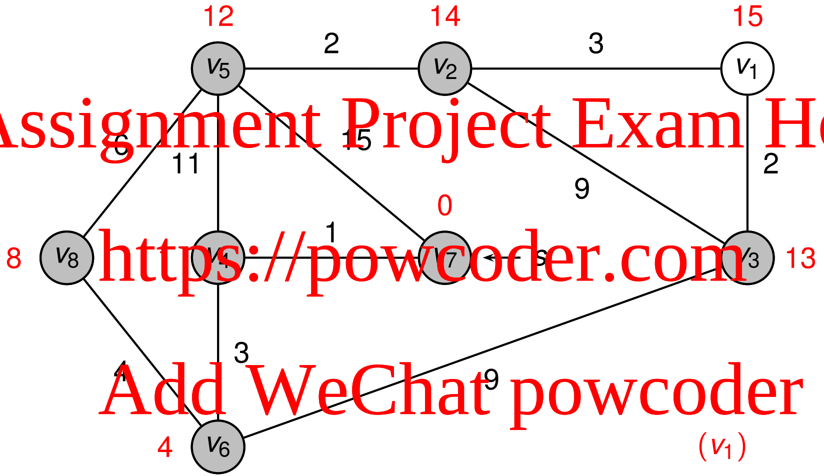
All vertices adjacent to v_3 now considered

Illustration of Dijkstra's Algorithm



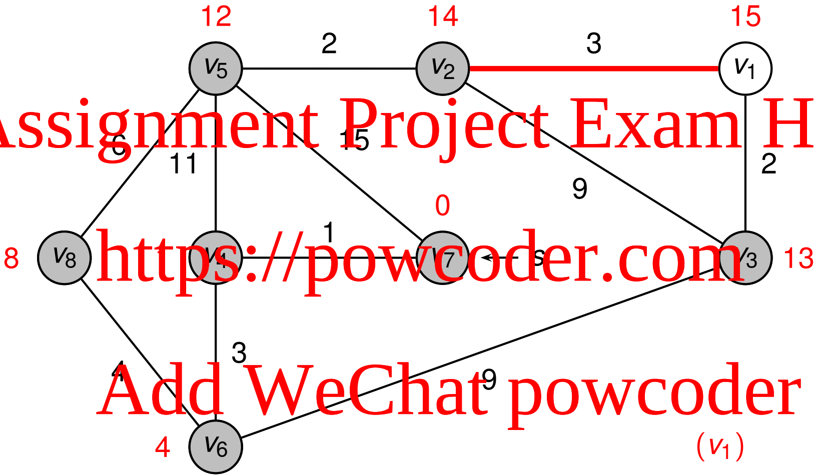
Remove v_2 from Q and add to A

Illustration of Dijkstra's Algorithm



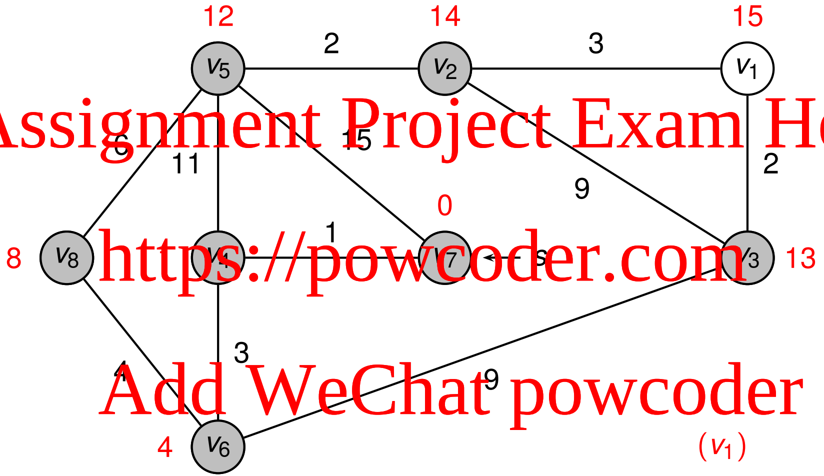
Consider edge $\{v_2, v_1\}$

Illustration of Dijkstra's Algorithm



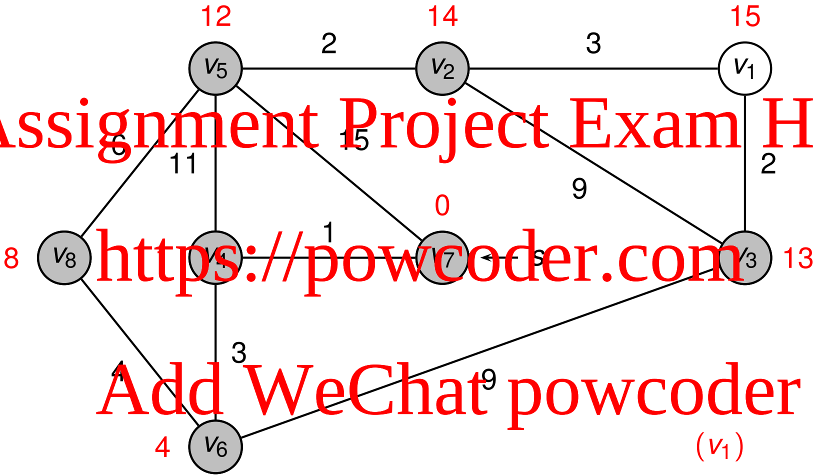
$\delta(v_1) < \delta(v_2) + w(v_2, v_1)$ so don't update $\delta(v_1)$

Illustration of Dijkstra’s Algorithm



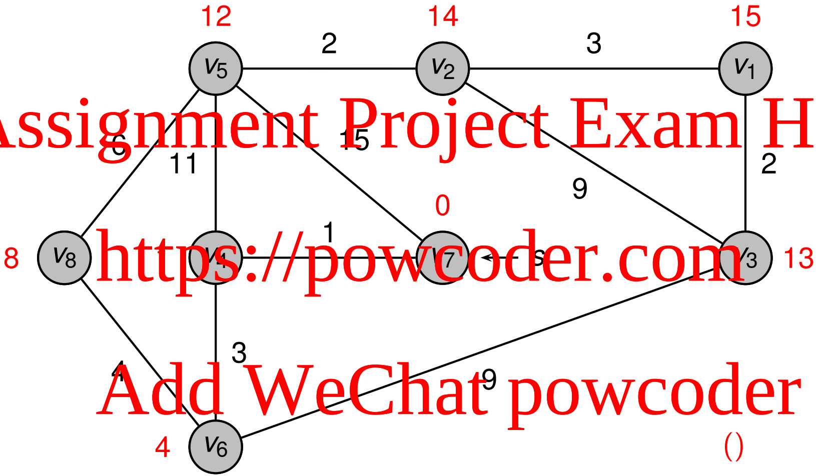
All vertices adjacent to v_2 now considered

Illustration of Dijkstra's Algorithm



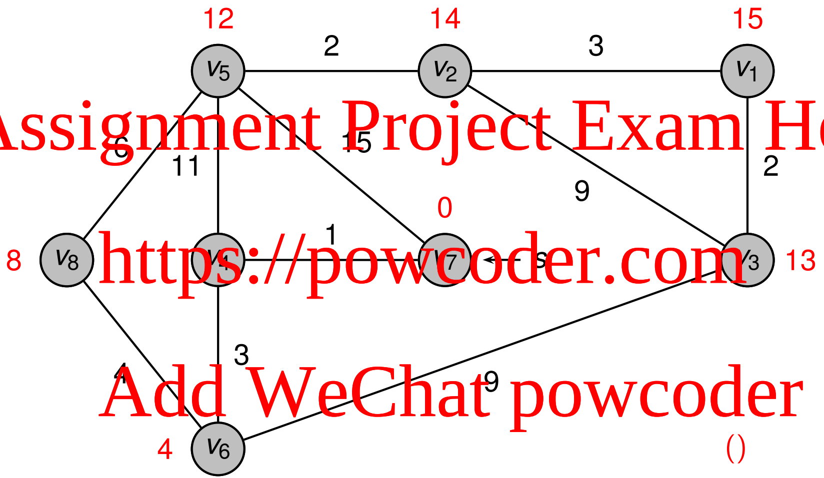
Remove v_1 from Q and add to A

Illustration of Dijkstra's Algorithm



No vertices adjacent to v_1 need to be considered

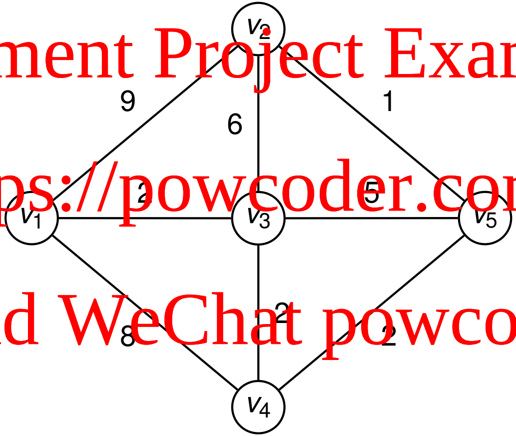
Illustration of Dijkstra's Algorithm



Priority queue Q is empty — all done!

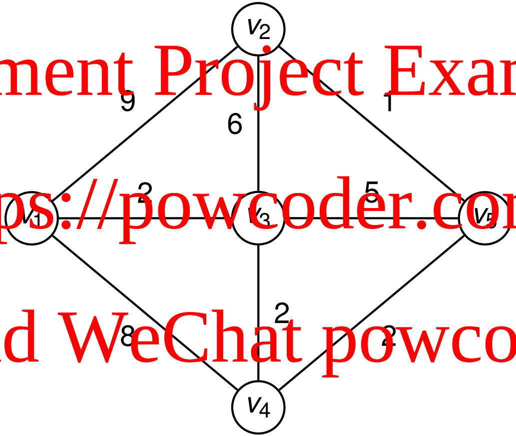
Problem for you

Run Dijkstra's Algorithm on this graph with $s = v_1$



Problem for you

Run Dijkstra's Algorithm on this graph with $s = v_1$



$\delta(v_1) = 0$, $\delta(v_2) = 7$, $\delta(v_3) = 2$, $\delta(v_4) = 4$, $\delta(v_5) = 6$

Proving Correctness of Dijkstra's Algorithm

Claim: As each v is added to A, $\delta(v)$ is correct distance of v from s

Proof by contradiction:

- Suppose that v is *first* vertex added to A with wrong $\delta(v)$
- Let p be a shortest path from s to v
- So we know that $w(p) < \delta(v)$

Proving Correctness of Dijkstra's Algorithm

Another Claim: p must include a vertex not yet in A

Proof by contraction:

- Suppose that p only includes vertices in A
- Let u be the vertex immediately before v on the path p
- $\delta(u)$ must be correct since v was first vertex with wrong value
- We would have had $\delta(v) = \delta(u) + w(u, v)$
- So $\delta(v)$ would then had correct value $w(p)$
- Yet later (when v added to A) $w(p) < \delta(v)$
- Impossible: value of $\delta(v)$ can't go up!

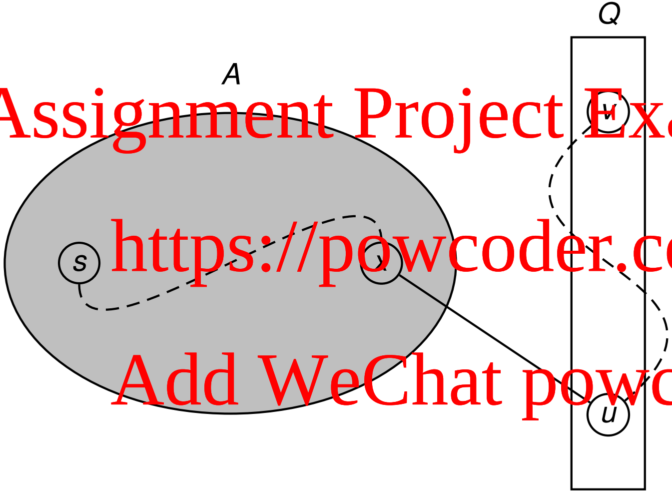
Proving Correctness of Dijkstra's Algorithm

Return to previous proof

- Established that p includes a vertex not in A
- Let u be the *first* vertex in p not in A
- Let x immediately precede u in p

Time for a picture

Proving Correctness of Dijkstra's Algorithm



Proving Correctness of Dijkstra's Algorithm

Continuing proof

- $\delta(x)$ must be correct
- When x was added to A we would have $\delta(u) = \delta(x) + w(x, u)$
- But $\delta(u) < w(p)$ because u appears inside path p
- So $\delta(u) < \delta(v)$ because we know that $w(p) < \delta(v)$
- So we would have added u not v to A
- Contradiction!

Dijkstra's Algorithm

Dijkstra(G,w,s):

let A be the empty set
let $\delta(s) = 0$
let $\delta(v) = \infty$ *for* $v \in V - \{s\}$
let Q be a priority queue containing elements of V
while Q is not empty
  remove v from front of priority queue Q
  add v to A
  for each $(v, u) \in E$ *where* $u \notin A$
    if $\delta(u) > \delta(v) + w(v, u)$ *then*
      let $\delta(u) = \delta(v) + w(v, u)$

Running time of Dijkstra's Algorithm

- Initialisation of Q takes $O(n)$ time
- Measure of progress is number of vertices in A
- Loop is executed n times
- Each iteration involves at most outdegree(v) updates of δ values
- Total of m updates of δ values over all iterations
- Each update of a δ value takes $O(\log n)$ steps
- Total running time is $O(m \log n)$
- Can also be written $O(n^2 \log n)$

Question for you

- What would the running time of Dijkstra's Algorithm be if the priority queue is implemented with an unsorted list?

Question for you

- What would the running time of Dijkstra's Algorithm be if the priority queue is implemented with an unsorted list?

There would be n iterations of while loop with $O(n)$ per iteration. So total running time would be $O(n^2)$.

Question for you

- Precisely specify the loop invariant of Dijsktra's Algorithm.
  In particular, specify the value $\delta(v)$ for vertices v that are still in Q.
  Recall that the set of vertices that are no longer in Q is denoted A.

Question for you

- Precisely specify the loop invariant of Dijsktra's Algorithm.
  In particular, specify the value $\delta(v)$ for vertices v that are still in Q.
  Recall that the set of vertices that are no longer in Q is denoted A.

See earlier slides