Aleks Ignjatov

School of Computer Science and En
University of New South Wales

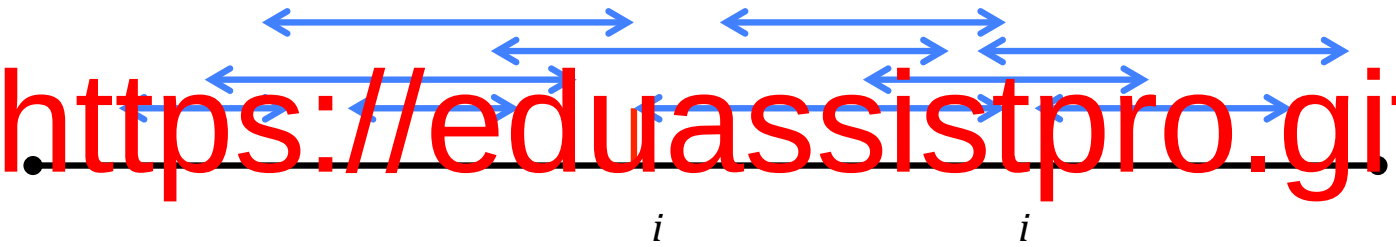
6. THE GREEDY METHOD

The Greedy Method

Activity selection problem.

Instance: A list of activities a_i, $(1 \leq i \leq n)$ with starting times s_i and finishing times f_i. No two activities can take place simultaneously.

Task: Find a *maximum size* subset of compatible activities.



Attempt 1: always choose the shortest activity whic previously chosen activities, remove the conflicting activi

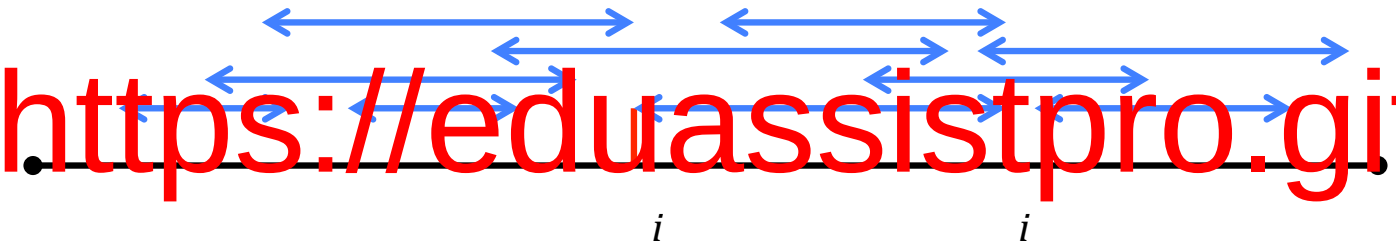
- The above figure shows this does not work...
  (chosen activities in green, conflicting in red)

The Greedy Method

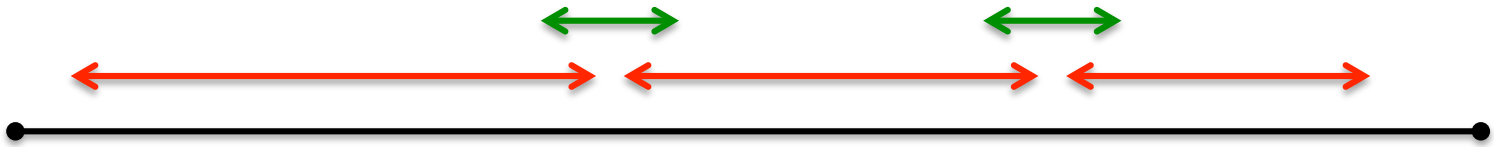
Activity selection problem.

Instance: A list of activities a_i, $(1 \leq i \leq n)$ with starting times s_i and finishing times f_i. No two activities can take place simultaneously.

Task: Find a *maximum size* subset of compatible activities.



Attempt 1: always choose the shortest activity whic previously chosen activities, remove the conflicting activi



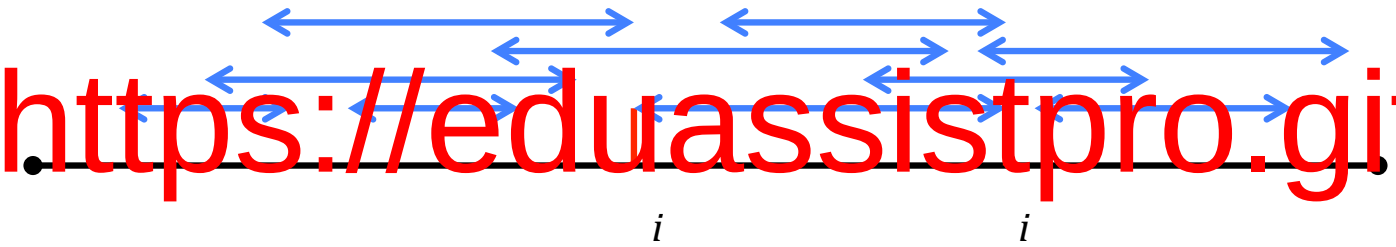
- The above figure shows this does not work...
  (chosen activities in green, conflicting in red)

The Greedy Method

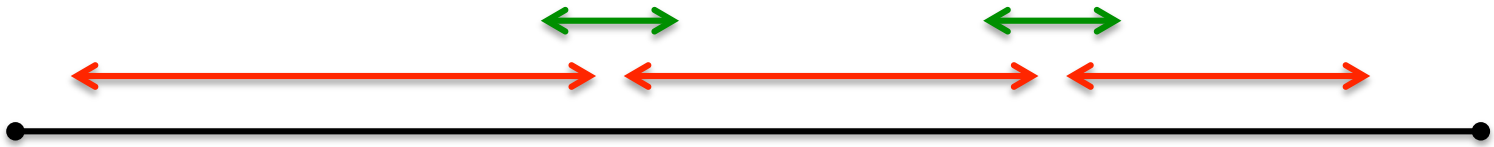
Activity selection problem.

Instance: A list of activities a_i, $(1 \leq i \leq n)$ with starting times s_i and finishing times f_i. No two activities can take place simultaneously.

Task: Find a *maximum size* subset of compatible activities.



Attempt 1: always choose the shortest activity whic previously chosen activities, remove the conflicting activi



- The above figure shows this does not work...
  (chosen activities in green, conflicting in red)

The Greedy Method

Activity selection problem.

Instance: A list of activities a_i, $(1 \leq i \leq n)$ with starting times s_i and finishing times f_i. No two activities can take place simultaneously.

Task: Find a *maximum size* subset of compatible activities.

- **Att**
  the fe
  this way we minimally restrict our next choice....

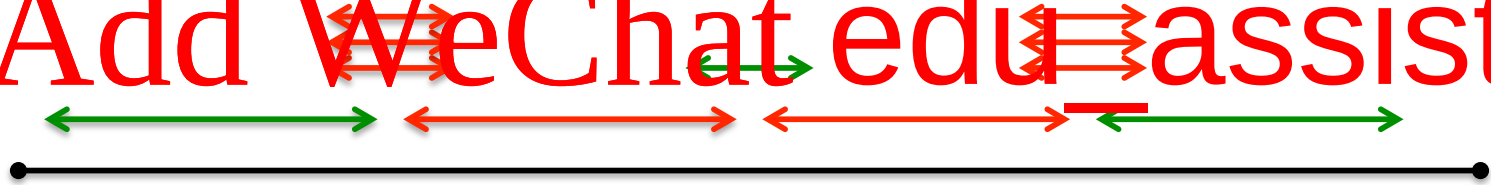
- As appealing this idea is, the above figure shows this again does not work ...

Activity selection problem.

Instance: A list of activities a_i, $(1 \leq i \leq n)$ with starting times s_i and finishing times f_i. No two activities can take place simultaneously.

Task: Find a *maximum size* subset of compatible activities.

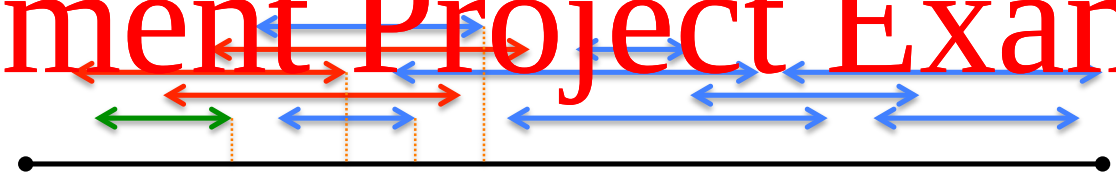
- **Att**
  the fe
  this way we minimally restrict our next choice....

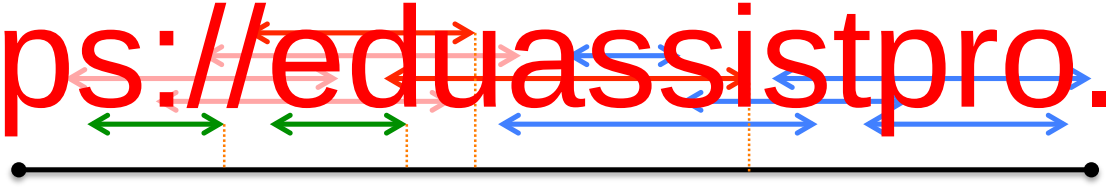


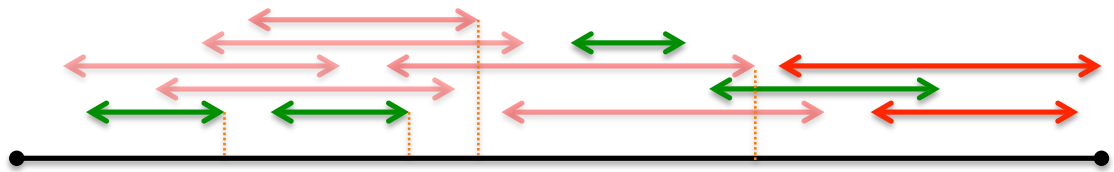
- As appealing this idea is, the above figure shows this again does not work ...

The Greedy Method

The correct solution: Among the activities which do not conflict with the previously chosen activities always chose the one with the earliest end time.



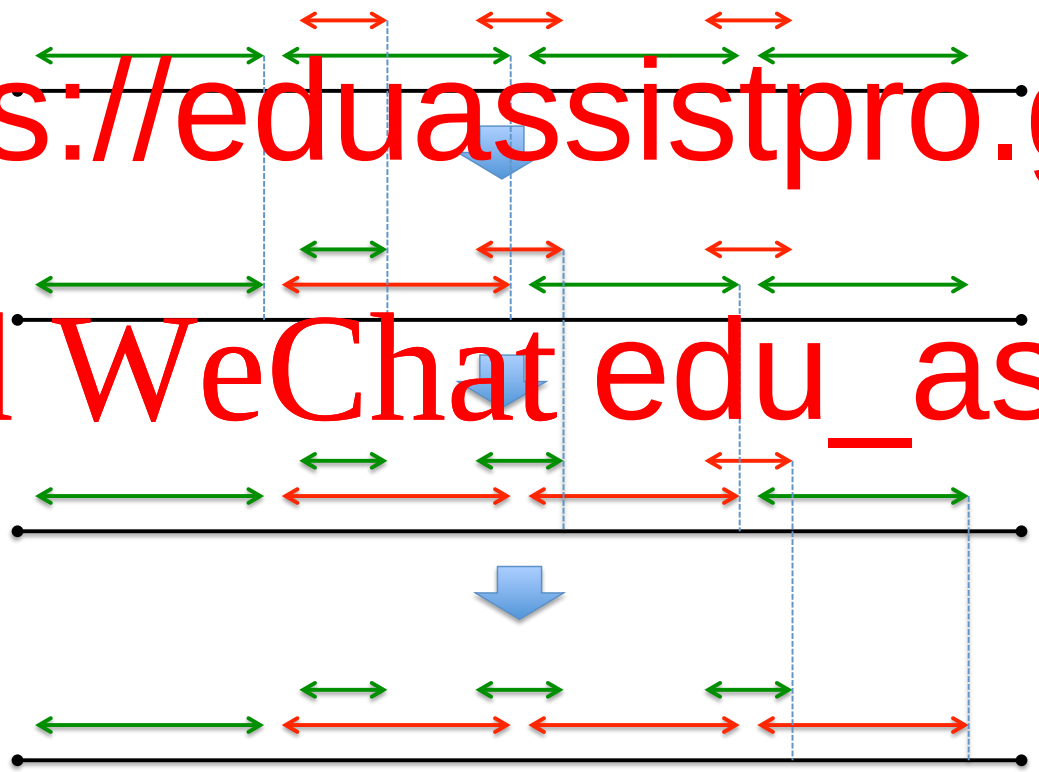




Proving optimality of a greedy solution

Show that any optimal solution can be transformed into the greedy solution with equal number of activities:

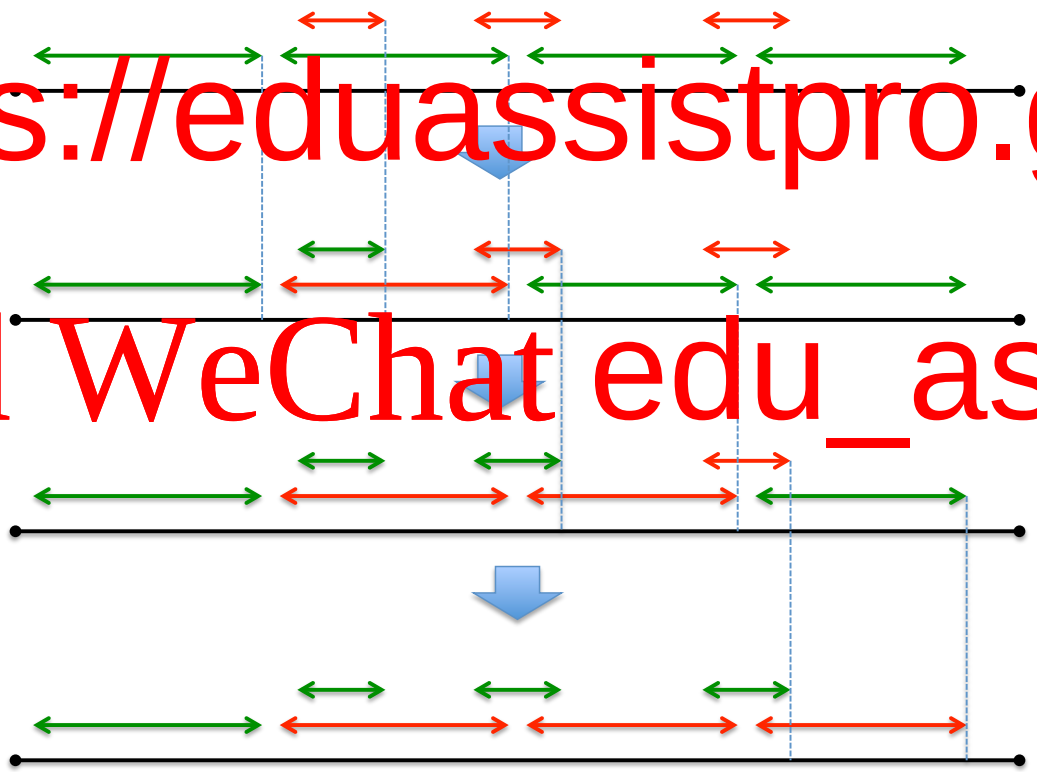
- Find the first place where the chosen activity violates the greedy choice.
- Show that replacing that activity with the greedy choice produces a non-conflicting solution with the same number of activities.
- Continue in this manner till you "morph" your optimal solution into the greedy solution, thus proving the greedy solution is also optimal.



Proving optimality of a greedy solution

Show that any optimal solution can be transformed into the greedy solution with equal number of activities:

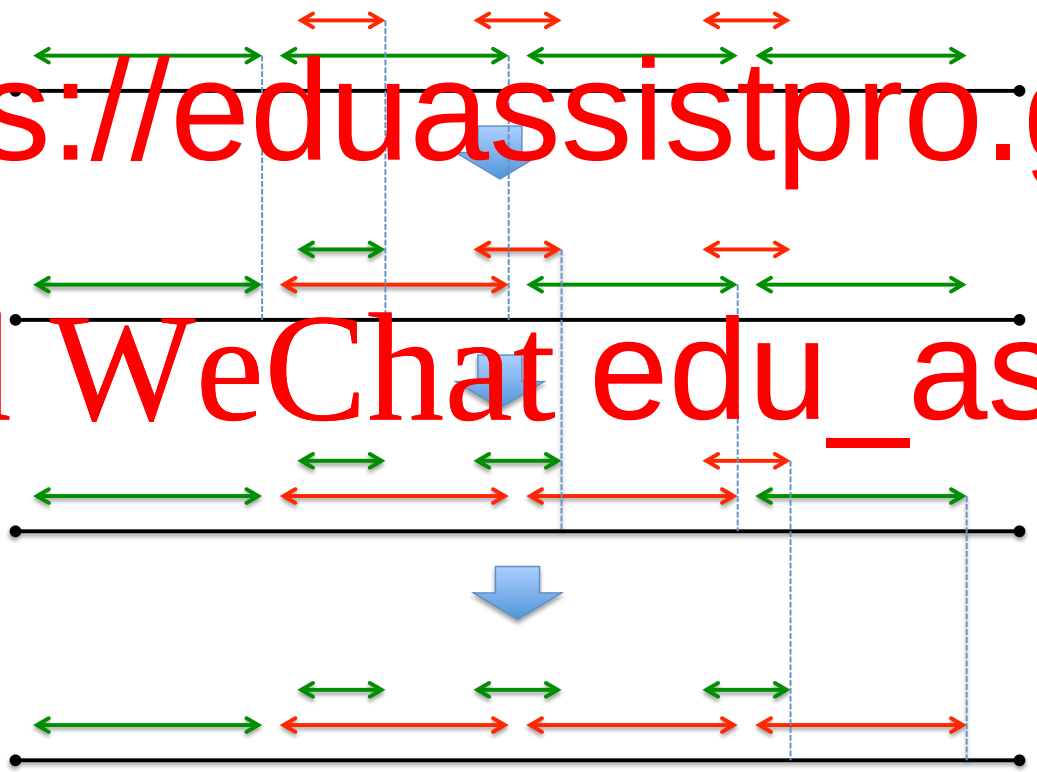
- Find the first place where the chosen activity violates the greedy choice.
- Show that replacing that activity with the greedy choice produces a non-conflicting selection with the same number of activities.
- Continue in this manner till you "morph" your optimal solution into the greedy solution, thus proving the greedy solution is also optimal.



Proving optimality of a greedy solution

Show that any optimal solution can be transformed into the greedy solution with equal number of activities:

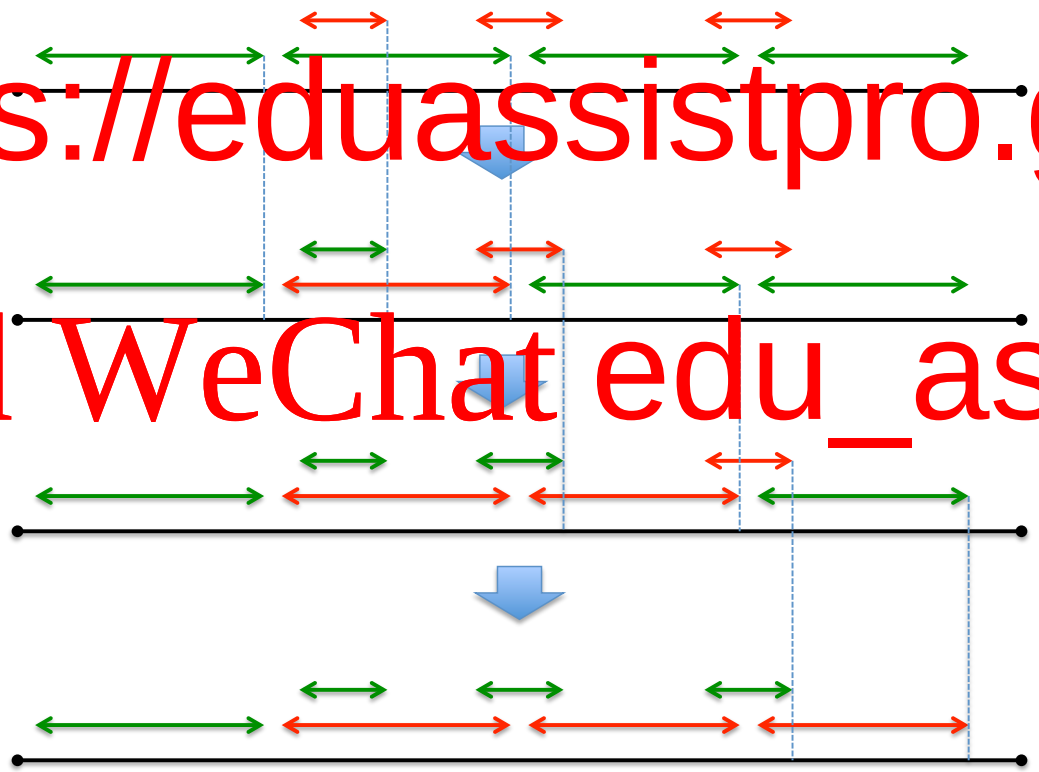
- Find the first place where the chosen activity violates the greedy choice.
- Show that replacing that activity with the greedy choice produces a non-conflicting selection with the same number of activities.
- Continue in this manner till you "morph" your optimal solution into the greedy solution, thus proving the greedy solution is also optimal.



Proving optimality of a greedy solution

Show that any optimal solution can be transformed into the greedy solution with equal number of activities:

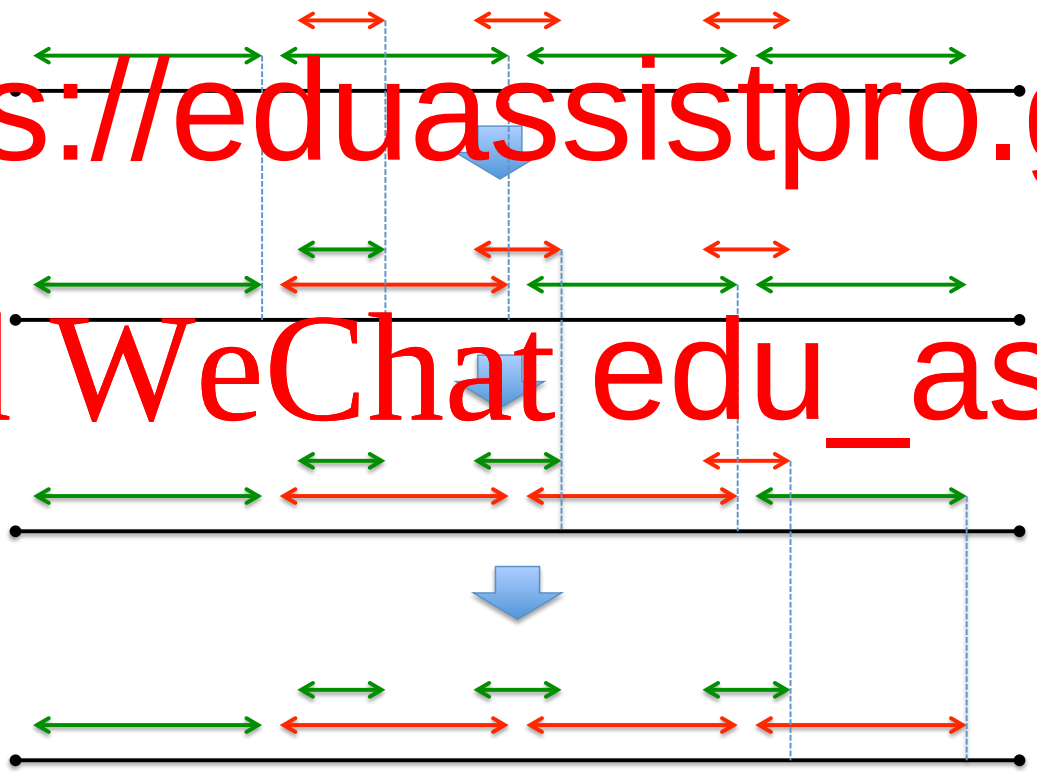
- Find the first place where the chosen activity violates the greedy choice.
- Show that replacing that activity with the greedy choice produces a non conflicting selection with the same number of activities.
- Continue in this manner till you "morph" your optimal solution into the greedy solution, thus proving the greedy solution is also optimal.



Proving optimality of a greedy solution

Show that any optimal solution can be transformed into the greedy solution with equal number of activities:

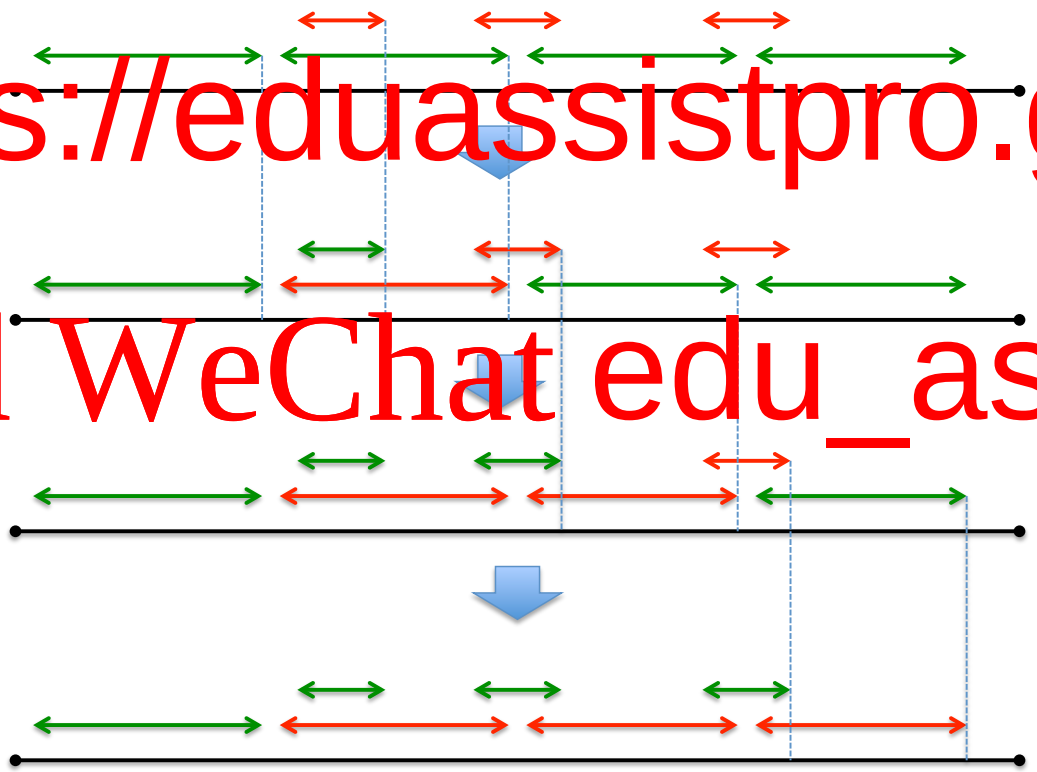
- Find the first place where the chosen activity violates the greedy choice.
- Show that replacing that activity with the greedy choice produces a non conflicting selection with the same number of activities.
- Continue in this manner till you "morph" your optimal solution into the greedy solution, thus proving the greedy solution is also optimal.



Proving optimality of a greedy solution

Show that any optimal solution can be transformed into the greedy solution with equal number of activities:

- Find the first place where the chosen activity violates the greedy choice.
- Show that replacing that activity with the greedy choice produces a non conflicting selection with the same number of activities.
- Continue in this manner till you "morph" your optimal solution into the greedy solution, thus proving the greedy solution is also optimal.



- What is the time complexity of the algorithm?

- We represent activities by ordered pairs of their starting and then finishing times and sort them in an increasing order of their finishing tim

- Thi

- We go through such a sorted list in order, looking for the first activity whose starting time is after the finishing time of the la

- Every activity is handled only once, so this part of th $O(n)$ time.

- Thus, the algorithm runs in total time $O(n \log n)$.

The Greedy Method

- What is the time complexity of the algorithm?
- We represent activities by ordered pairs of their starting and their finishing times and sort them in an increasing order of their finishing tim
- Thi
- We go through such a sorted list in order, looking for the first activity whose starting time is after the finishing time of the la
- Every activity is handled only once, so this part of th $O(n)$ time.
- Thus, the algorithm runs in total time $O(n \log n)$.

The Greedy Method

- What is the time complexity of the algorithm?
- We represent activities by ordered pairs of their starting and their finishing times and sort them in an increasing order of their finishing tim
- Thi
- We go through such a sorted list in order, looking for the first activity whose starting time is after the finishing time of the la
- Every activity is handled only once, so this part of th $O(n)$ time.
- Thus, the algorithm runs in total time $O(n \log n)$.

The Greedy Method

- What is the time complexity of the algorithm?
- We represent activities by ordered pairs of their starting and their finishing times and sort them in an increasing order of their finishing tim
- Thi
- We go through such a sorted list in order, looking for the first activity whose starting time is after the finishing time of the la
- Every activity is handled only once, so this part of th $O(n)$ time.
- Thus, the algorithm runs in total time $O(n \log n)$.

The Greedy Method

- What is the time complexity of the algorithm?
- We represent activities by ordered pairs of their starting and their finishing times and sort them in an increasing order of their finishing tim
- Thi
- We go through such a sorted list in order, looking for the first activity whose starting time is after the finishing time of the la
- Every activity is handled only once, so this part of th $O(n)$ time.
- Thus, the algorithm runs in total time $O(n \log n)$.

The Greedy Method

- What is the time complexity of the algorithm?
- We represent activities by ordered pairs of their starting and their finishing times and sort them in an increasing order of their finishing tim
- Thi
- We go through such a sorted list in order, looking for the first activity whose starting time is after the finishing time of the la
- Every activity is handled only once, so this part of th $O(n)$ time.
- Thus, the algorithm runs in total time $O(n \log n)$.

Activity selection problem II

- **Instance:** A list of activities a_i, $(1 \leq i \leq n)$ with starting times s_i and finishing times $f_i = s_i + d_i$ thus, all activities are of the same duration. No two activities can take place simultaneously.
- Ta
- Sol
to fin
i.e., the previous problem.
- **Question:** What happens if the activities are n
duration?
- Greedy strategy no longer works - we will need a more sophisticated technique.

Activity selection problem II

- **Instance:** A list of activities a_i, $(1 \leq i \leq n)$ with starting times s_i and finishing times $f_i = s_i + d_i$ thus, all activities are of the same duration. No two activities can take place simultaneously.

- **Ta** *uration.*

- **Sol**
  to fin
  i.e., the previous problem.

- **Question:** What happens if the activities are n
  duration?

- Greedy strategy no longer works - we will need a more sophisticated technique.

Activity selection problem II

- **Instance:** A list of activities a_i, $(1 \leq i \leq n)$ with starting times s_i and finishing times $f_i = s_i + d_i$ thus, all activities are of the same duration. No two activities can take place simultaneously.
- **Ta** *uration.*
- **Sol** to fin i.e., the previous problem.

- **Question:** What happens if the activities are n duration?
- Greedy strategy no longer works - we will need a more sophisticated technique.

Activity selection problem II

- **Instance:** A list of activities a_i, $(1 \leq i \leq n)$ with starting times s_i and finishing times $f_i = s_i + d_i$ thus, all activities are of the same duration. No two activities can take place simultaneously.

- **Ta** *uration.*

- **Sol**
to fin
i.e., the previous problem.

- **Question:** What happens if the activities are n duration?

- Greedy strategy no longer works - we will need a more sophisticated technique.

Activity selection problem II

- **Instance:** A list of activities a_i, $(1 \leq i \leq n)$ with starting times s_i and finishing times $f_i = s_i + d_i$ thus, all activities are of the same duration. No two activities can take place simultaneously.

- **Ta** *uration.*

- **Sol**
to fin
i.e., the previous problem.

- **Question:** What happens if the activities are n duration?

- Greedy strategy no longer works - we will need a more sophisticated technique.

More practice problems for the Greedy Method

- Along the long, straight road from Loololong to Goolagong houses are scattered quite sparsely, sometimes with long gaps between two consecutive houses. Telstra must provide mobile phone service to people who live alongside the road, and the range of Telstra's cell base station is 4km. Design an algorithm for placing the minimal number of base stations alongside the road, that is sufficient to cover all houses.

More practice problems for the Greedy Method

- Once again, along the long, straight road from Loololong (on the West) to Goolagong (on the East) houses are scattered quite sparsely, sometimes with long gaps between two consecutive houses. Telstra must provide mobile phone service to people who live alongside the road, and the range of Telstras cell base station is 5km.
- One put not the E
- His junior associate did exactly the same but starti moving westwards and claimed that is the best!
- Is there a placement of houses for which the associate is right?

More practice problems for the Greedy Method

- Once again, along the long, straight road from Loololong (on the West) to Goolagong (on the East) houses are scattered quite sparsely, sometimes with long gaps between two consecutive houses. Telstra must provide mobile phone service to people who live alongside the road, and the range of Telstras cell base station is 5km.

- One
put a
not a
the E

- His junior associate did exactly the same but starti
moving westwards and claimed that is the best.

- Is there a placement of houses for which the associate is right?

More practice problems for the Greedy Method

- Once again, along the long, straight road from Loololong (on the West) to Goolagong (on the East) houses are scattered quite sparsely, sometimes with long gaps between two consecutive houses. Telstra must provide mobile phone service to people who live alongside the road, and the range of Telstras cell base station is 5km.

- One
put a
not a
the E

- His junior associate did exactly the same but starti
moving westwards and claimed that his method r

- Is there a placement of houses for which the associate is right?

More practice problems for the Greedy Method

- Once again, along the long, straight road from Loololong (on the West) to Goolagong (on the East) houses are scattered quite sparsely, sometimes with long gaps between two consecutive houses. Telstra must provide mobile phone service to people who live alongside the road, and the range of Telstras cell base station is 5km.

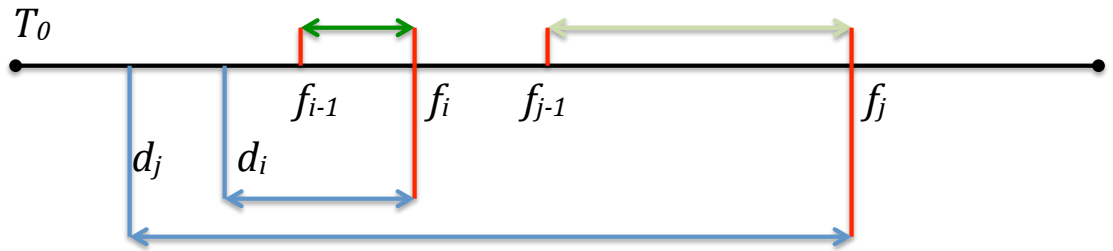
- One
put a
not a
the E

- His junior associate did exactly the same but starti
moving westwards and claimed that his method r

- Is there a placement of houses for which the associate is right?

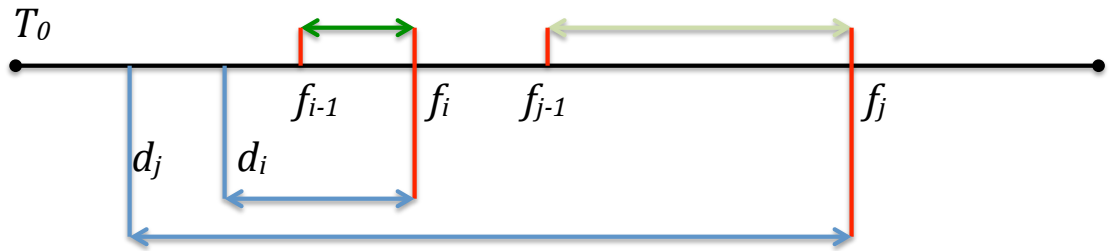
Minimising job lateness

- **Instance:** A start time T_0 and a list of jobs a_i, $(1 \leq i \leq n)$, with duration times t_i and deadlines d_i. Only one job can be performed at any time all jobs have to be completed. If a job a_i is completed at a finishing time $f_i > d_i$ then we say that it has incurred lateness $l_i = f_i - d_i$.
- **Tas**
late
- **Sol**
deadlines.
- **Optimality:** Consider any optimal solution. W bs a_j form an inversion if a_i is scheduled before



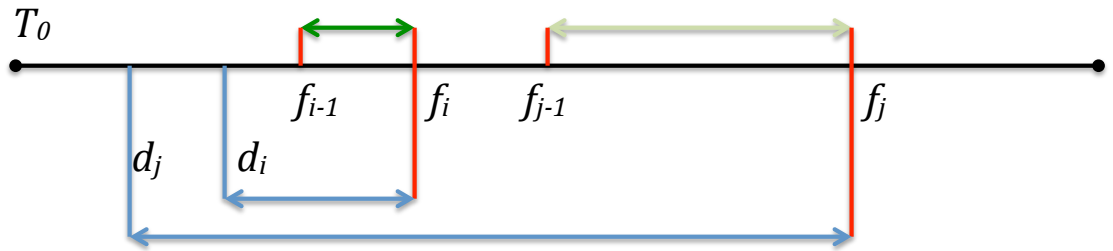
Minimising job lateness

- **Instance:** A start time T_0 and a list of jobs a_i, $(1 \leq i \leq n)$, with duration times t_i and deadlines d_i. Only one job can be performed at any time, all jobs have to be completed. If a job a_i is completed at a finishing time $f_i > d_i$ then we say that it has incurred lateness $l_i = f_i - d_i$.
- **Tas**
  late
- **Sol**
  deadlines.
- **Optimality:** Consider any optimal solution. W… …bs a_j … form an inversion if a_i is scheduled before …



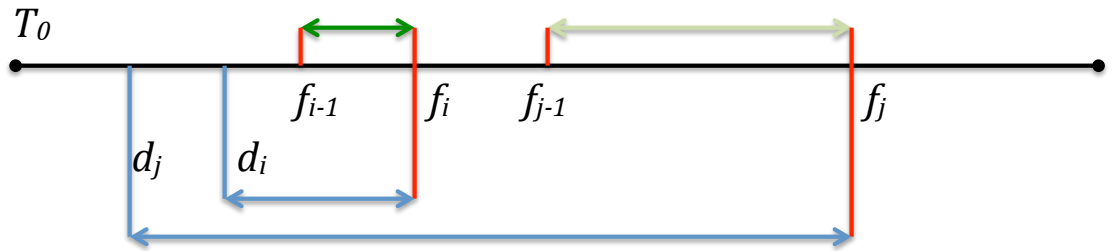
Minimising job lateness

- **Instance:** A start time T_0 and a list of jobs a_i, $(1 \leq i \leq n)$, with duration times t_i and deadlines d_i. Only one job can be performed at any time, all jobs have to be completed. If a job a_i is completed at a finishing time $f_i > d_i$ then we say that it has incurred lateness $l_i = f_i - d_i$.
- **Tas**
late
- **Sol**
deadlines.
- **Optimality:** Consider any optimal solution. W bs a_j form an inversion if a_i is scheduled before



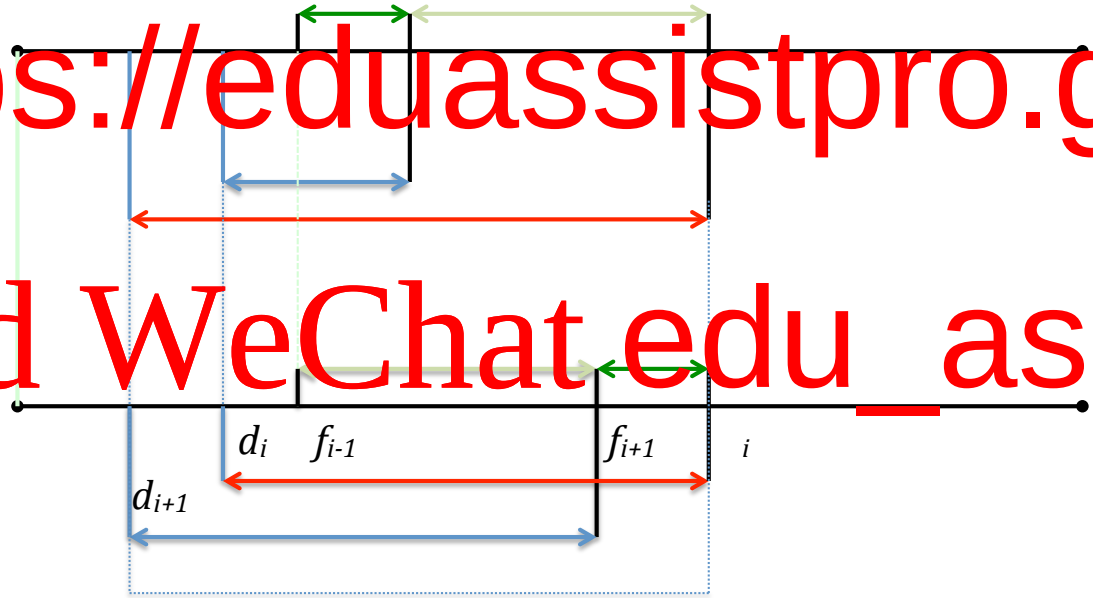
Minimising job lateness

- **Instance:** A start time T_0 and a list of jobs a_i, $(1 \leq i \leq n)$, with duration times t_i and deadlines d_i. Only one job can be performed at any time, all jobs have to be completed. If a job a_i is completed at a finishing time $f_i > d_i$ then we say that it has incurred lateness $l_i = f_i - d_i$.
- **Tas**
  late
- **Sol**
  deadlines.
- **Optimality:** Consider any optimal solution. W bs a_j form an inversion if job a_j is scheduled before a_i



Minimising job lateness

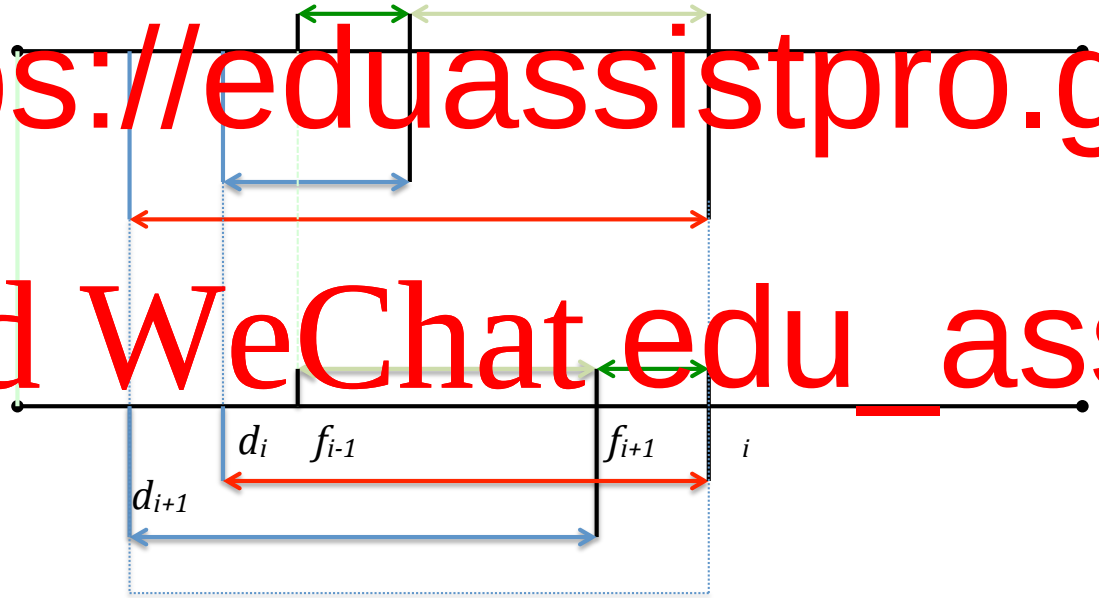
- We will show that there exists a scheduling without inversions which is also optimal.
- Recall the `BubbleSort`: if we manage to eliminate all inversions between adjacent jobs, eventually all the inversions will be eliminated.



- Note that swapping adjacent inverted jobs reduces the larger lateness!

Minimising job lateness

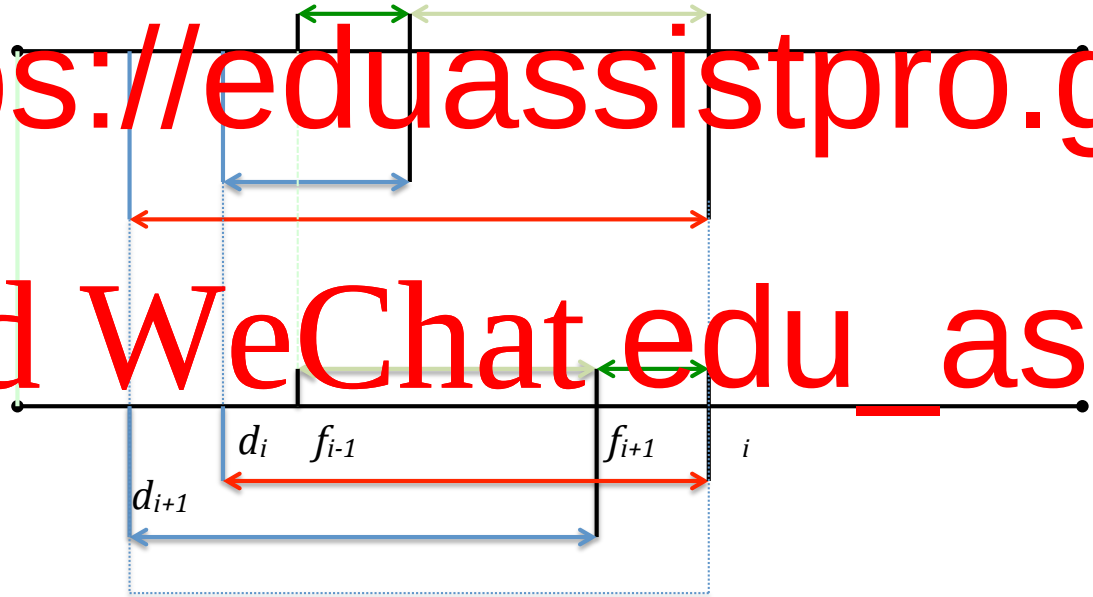
- We will show that there exists a scheduling without inversions which is also optimal.
- Recall the `BubbleSort`: if we manage to eliminate all inversions between adjacent jobs, eventually all the inversions will be eliminated.



- Note that swapping adjacent inverted jobs reduces the larger lateness!

Minimising job lateness

- We will show that there exists a scheduling without inversions which is also optimal.
- Recall the `BubbleSort`: if we manage to eliminate all inversions between adjacent jobs, eventually all the inversions will be eliminated.



- Note that swapping adjacent inverted jobs reduces the larger lateness!

More practice problems for the Greedy Method

Tape storage

- **Instance:** A list of n files f_i of lengths l_i which have to be stored on a tape. Each file is equally likely to be needed. To retrieve a file, one must start from the beginning of the tape and scan it until the file is found and read.

- **Tas**
  is mi

- **Solution:** If the files are stored in order $l_1, l_2, \ldots l_n$, then the expected time is proportional to

$$nl_1 + (n-1)l_2 + \qquad 3 \qquad n-1 \quad n$$

- This is minimised if $l_1 \leq l_2 \leq l_3 \leq \ldots \leq l_n$.

More practice problems for the Greedy Method

Tape storage

- **Instance:** A list of n files f_i of lengths l_i which have to be stored on a tape. Each file is equally likely to be needed. To retrieve a file, one must start from the beginning of the tape and scan it until the file is found and read.

- **Tas**
  is min

- **Solution:** If the files are stored in order $l_1, l_2, \ldots l_n$, then the expected time is proportional to

$$l_1 + (l_1 + l_2) + (l_1 + l_2 + l_3) + \ldots$$
$$nl_1 + (n-1)l_2 + \qquad 3 \qquad n-1 \quad n$$

- This is minimised if $l_1 \leq l_2 \leq l_3 \leq \ldots \leq l_n$.

More practice problems for the Greedy Method

Tape storage

- **Instance:** A list of n files f_i of lengths l_i which have to be stored on a tape. Each file is equally likely to be needed. To retrieve a file, one must start from the beginning of the tape and scan it until the file is found and read.

- **Tas**
  is min

- **Solution:** If the files are stored in order $l_1, l_2, \ldots l_n$, then the expected time is proportional to

$$l_1 + (l_1 + l_2) + (l_1 + l_2 + l_3) + \ldots$$
$$nl_1 + (n-1)l_2 + \qquad 3 \qquad n-1 \quad n$$

- This is minimised if $l_1 \leq l_2 \leq l_3 \leq \ldots \leq l_n$.

More practice problems for the Greedy Method

Tape storage

- **Instance:** A list of n files f_i of lengths l_i which have to be stored on a tape. Each file is equally likely to be needed. To retrieve a file, one must start from the beginning of the tape and scan it until the file is found and read.

- **Tas**
  is min

- **Solution:** If the files are stored in order $l_1, l_2, \ldots l_n$, then the expected time is proportional to

$$l_1 + (l_1 + l_2) + (l_1 + l_2 + l_3) + \ldots$$
$$nl_1 + (n-1)l_2 + \qquad 3 \qquad n-1 \quad n$$

- This is minimised if $l_1 \leq l_2 \leq l_3 \leq \ldots \leq l_n$.

Tape storage II

- **Instance:** A list of n files f_i of lengths l_i and probabilities to be needed p_i, $\sum_{i=1}^{n} p_i = 1$, which have to be stored on a tape. To retrieve a file, one must start from the beginning of the tape and scan it until the file is fund and read.
- **Tas** l ime is mini
- **Sol** 1 2 n d time is proportional to
- We now show that this is minimised if the files are ordered in a decreasing order of values of the ratio p_i/l_i.

Tape storage II

- **Instance:** A list of n files f_i of lengths l_i and probabilities to be needed p_i, $\sum_{i=1}^{n} p_i = 1$, which have to be stored on a tape. To retrieve a file, one must start from the beginning of the tape and scan it until the file is fund and read.
- **Tas** l ime is mini
- **Sol** d time is proportional to $1 \quad 2 \quad n$

  $p_1 l_1 + (l_1 + l_2)p_2 + \cdots + (l_1 + l_2 + \cdots + l_n)p_n$
- We now show that this is minimised if the files are ordered in a decreasing order of values of the ratio p_i/l_i.

Tape storage II

- **Instance:** A list of n files f_i of lengths l_i and probabilities to be needed p_i, $\sum_{i=1}^{n} p_i = 1$ which have to be stored on a tape. To retrieve a file, one must start from the beginning of the tape and scan it until the file is fund and read.
- **Tas** l ime is mini
- **Sol** 1 2 n d time is proportional to

$$p_1 l_1 + (l_1 + l_2) p_2 + (l_1 + l_2 + l_3) p_3 + \ldots p_n$$

- We now show that this is minimised if the files are ordered in a decreasing order of values of the ratio p_i/l_i.

Tape storage II

- **Instance:** A list of n files f_i of lengths l_i and probabilities to be needed p_i, $\sum_{i=1}^{n} p_i = 1$ which have to be stored on a tape. To retrieve a file, one must start from the beginning of the tape and scan it until the file is fund and read.
- **Tas** l ime is mini
- **Sol** 1 2 n d time is proportional to

$$p_1 l_1 + (l_1 + l_2) p_2 + (l_1 + l_2 + l_3) p_3 + \ldots p_n$$

- We now show that this is minimised if the files are ordered in a decreasing order of values of the ratio p_i/l_i.

More practice problems for the Greedy Method

- Let us see what happens if we swap to adjacent files f_k and f_{k+1}.
- The expected time before the swap and after the swap are, respectively,

$$E = l_1 p_1 + (l_1 + l_2)p_2 + (l_1 + l_2 + l_3)p_3 + \ldots + (l_1 + l_2 + l_3 + \ldots + l_{k-1} + l_k)p_k + (l_1 + l_2 + l_3 + \ldots + l_{k-1} + l_k + l_{k+1})p_{k+1} + \ldots + (l_1 + l_2 + l_3 + \ldots + l_n)p_n$$

and

$$E' = l_1 p_1 + (l_1 + l_2)p_2 + (l_1 + l_2 + l_3)p_3 + \ldots + (l_1 + l_2 + l_3 + \ldots + l_{k-1} + l_{k+1})p_{k+1} + (l_1 + l_2 + l_3 + \ldots + l_{k-1} + l_{k+1} + l_k)p_k + \ldots + (l_1 + l_2 + l_3 + \ldots + l_n)p_n$$

- Thus, $E - E' = l_k p_{k+1} - l_{k+1} p_k$, which is posit $l_k p_{k+1} > l_{k+1} p_k$, i.e., if $p_k/l_k < p_{k+1}/l_{k+1}$.
- Consequently, $E > E'$ if and only if the swap decreases the expected time just in case there is an inversion: a file f_{k+1} with a larger ratio p_{k+1}/l_{k+1} has been put after a file f_k with a smaller ratio p_k/l_k.
- For as long as there are inversions there will be inversions of consecutive files and swapping will reduce the expected time. Consequently, the optimal solution is the one with no inversions.

More practice problems for the Greedy Method

- Let us see what happens if we swap to adjacent files f_k and f_{k+1}.
- The expected time before the swap and after the swap are, respectively,

$$E = l_1 p_1 + (l_1 + l_2)p_2 + (l_1 + l_2 + l_3)p_3 + \ldots + (l_1 + l_2 + l_3 + \ldots + l_{k-1} + l_k)p_k + (l_1 + l_2 + l_3 + \ldots + l_{k-1} + l_k + l_{k+1})p_{k+1} + \ldots + (l_1 + l_2 + l_3 + \ldots + l_n)p_n$$

and

$$E' = l_1 p_1 + (l_1 + l_2)p_2 + (l_1 + l_2 + l_3)p_3 + \ldots + (l_1 + l_2 + l_3 + \ldots + l_{k-1} + l_{k+1})p_k + (l_1 + l_2 + l_3 + \ldots + l_{k-1} + l_{k+1} + l_k)p_{k+1} + \ldots + (l_1 + l_2 + l_3 + \ldots + l_n)p_n$$

- Thus, $E - E' = l_k p_{k+1} - l_{k+1} p_k$, which is posit $l_k p_{k+1} > l_{k+1} p_k$, i.e., if $p_k/l_k < p_{k+1}/l_{k+1}$.
- Consequently, $E > E'$ if and only if the swap decreases the expected time just in case there is an inversion: a file f_{k+1} with a larger ratio p_{k+1}/l_{k+1} has been put after a file f_k with a smaller ratio p_k/l_k.
- For as long as there are inversions there will be inversions of consecutive files and swapping will reduce the expected time. Consequently, the optimal solution is the one with no inversions.

More practice problems for the Greedy Method

- Let us see what happens if we swap to adjacent files f_k and f_{k+1}.
- The expected time before the swap and after the swap are, respectively,

$$E = l_1 p_1 + (l_1 + l_2)p_2 + (l_1 + l_2 + l_3)p_3 + \ldots + (l_1 + l_2 + l_3 + \ldots + l_{k-1} + l_k)p_k + (l_1 + l_2 + l_3 + \ldots + l_{k-1} + l_k + l_{k+1})p_{k+1} + \ldots + (l_1 + l_2 + l_3 + \ldots + l_n)p_n$$

and

$$E' = l_1 p_1 + (l_1 + l_2)p_2 + (l_1 + l_2 + l_3)p_3 + \ldots + (l_1 + l_2 + l_3 + \ldots + l_{k-1} + l_{k+1})p_{k+1} + \ldots + (l_1 + l_2 + l_3 + \ldots + l_n)p_n$$

- Thus, $E - E' = l_k p_{k+1} - l_{k+1} p_k$, which is positive if $l_k p_{k+1} > l_{k+1} p_k$, i.e., if $p_k/l_k < p_{k+1}/l_{k+1}$.
- Consequently, $E > E'$ if and only if the swap decreases the expected time just in case there is an inversion: a file f_{k+1} with a larger ratio p_{k+1}/l_{k+1} has been put after a file f_k with a smaller ratio p_k/l_k.
- For as long as there are inversions there will be inversions of consecutive files and swapping will reduce the expected time. Consequently, the optimal solution is the one with no inversions.

More practice problems for the Greedy Method

- Let us see what happens if we swap to adjacent files f_k and f_{k+1}.
- The expected time before the swap and after the swap are, respectively,

$$E = l_1 p_1 + (l_1 + l_2)p_2 + (l_1 + l_2 + l_3)p_3 + \ldots + (l_1 + l_2 + l_3 + \ldots + l_{k-1} + l_k)p_k + (l_1 + l_2 + l_3 + \ldots + l_{k-1} + l_k + l_{k+1})p_{k+1} + \ldots + (l_1 + l_2 + l_3 + \ldots + l_n)p_n$$

and

$$E' = l_1 p_1 + (l_1 + l_2)p_2 + (l_1 + l_2 + l_3)p_3 + \ldots + (l_1 + l_2 + l_3 + \ldots + l_{k-1} + l_{k+1})p_{k+1} + \ldots + (l_1 + l_2 + l_3 + \ldots + l_n)p_n$$

- Thus, $E - E' = l_k p_{k+1} - l_{k+1} p_k$, which is positive if $l_k p_{k+1} > l_{k+1} p_k$, i.e., if $p_k/l_k < p_{k+1}/l_{k+1}$.
- Consequently, $E > E'$ if and only if the swap decreases the expected time just in case there is an inversion: a file f_{k+1} with a larger ratio p_{k+1}/l_{k+1} has been put after a file f_k with a smaller ratio p_k/l_k.
- For as long as there are inversions there will be inversions of consecutive files and swapping will reduce the expected time. Consequently, the optimal solution is the one with no inversions.

More practice problems for the Greedy Method

- Let us see what happens if we swap to adjacent files f_k and f_{k+1}.
- The expected time before the swap and after the swap are, respectively,

$$E = l_1 p_1 + (l_1 + l_2)p_2 + (l_1 + l_2 + l_3)p_3 + \ldots + (l_1 + l_2 + l_3 + \ldots + l_{k-1} + l_k)p_k + (l_1 + l_2 + l_3 + \ldots + l_{k-1} + l_k + l_{k+1})p_{k+1} + \ldots + (l_1 + l_2 + l_3 + \ldots + l_n)p_n$$

and

$$E' = l_1 p + (l_1 \quad -1 + l_{k+1})p_{k+1} + \ldots + n)p_n$$

- Thus, $E - E' = l_k p_{k+1} - l_{k+1} p_k$, which is posit $l_k p_{k+1} > l_{k+1} p_k$, i.e., if $p_k/l_k < p_{k+1}/l_{k+1}$.

- Consequently, $E > E'$ if and only if p_k/l_k e swap decreases the expected time just in case re is an inversion: a file f_{k+1} with a larger ratio p_{k+1}/l_{k+1} has been put after a file f_k with a smaller ratio p_k/l_k.

- For as long as there are inversions there will be inversions of consecutive files and swapping will reduce the expected time. Consequently, the optimal solution is the one with no inversions.

More practice problems for the Greedy Method

- Let us see what happens if we swap to adjacent files f_k and f_{k+1}.
- The expected time before the swap and after the swap are, respectively,

$$E = l_1 p_1 + (l_1 + l_2)p_2 + (l_1 + l_2 + l_3)p_3 + \ldots + (l_1 + l_2 + l_3 + \ldots + l_{k-1} + l_k)p_k + (l_1 + l_2 + l_3 + \ldots + l_{k-1} + l_k + l_{k+1})p_{k+1} + \ldots + (l_1 + l_2 + l_3 + \ldots + l_n)p_n$$

and

$$E' = l_1 p + (l_1 \quad -1 + l_{k+1})p_{k+1} + \ldots + n)p_n$$

- Thus, $E - E' = l_k p_{k+1} - l_{k+1} p_k$, which is posit $l_k p_{k+1} > l_{k+1} p_k$, i.e., if $p_k/l_k < p_{k+1}/l_{k+1}$.

- Consequently, $E > E'$ if and only if p_k/l_k e swap decreases the expected time just in case re is an inversion: a file f_{k+1} with a larger ratio p_{k+1}/l_{k+1} has been put after a file f_k with a smaller ratio p_k/l_k.

- For as long as there are inversions there will be inversions of consecutive files and swapping will reduce the expected time. Consequently, the optimal solution is the one with no inversions.

More practice problems for the Greedy Method

- Let us see what happens if we swap to adjacent files f_k and f_{k+1}.
- The expected time before the swap and after the swap are, respectively,

$$E = l_1 p_1 + (l_1 + l_2)p_2 + (l_1 + l_2 + l_3)p_3 + \ldots + (l_1 + l_2 + l_3 + \ldots + l_{k-1} + l_k)p_k + (l_1 + l_2 + l_3 + \ldots + l_{k-1} + l_k + l_{k+1})p_{k+1} + \ldots + (l_1 + l_2 + l_3 + \ldots + l_n)p_n$$

and

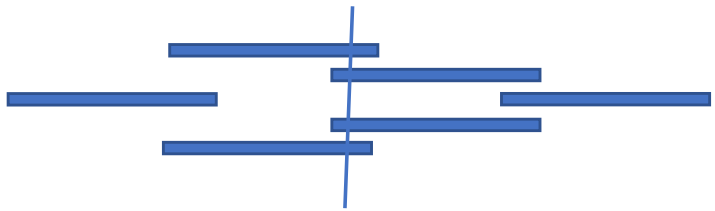
$$E' = l_1 p_1 + (l_1 + l_2)p_2 + (l_1 + l_2 + l_3)p_3 + \ldots + (l_1 + l_2 + l_3 + \ldots + l_{k-1} + l_{k+1})p_{k+1} + \ldots + (l_1 + l_2 + l_3 + \ldots + l_n)p_n$$

- Thus, $E - E' = l_k p_{k+1} - l_{k+1} p_k$, which is positive if $l_k p_{k+1} > l_{k+1} p_k$, i.e., if $p_k/l_k < p_{k+1}/l_{k+1}$.
- Consequently, $E > E'$ if and only if $p_k/l_k < p_{k+1}/l_{k+1}$. Therefore, a swap decreases the expected time just in case there is an inversion: a file f_{k+1} with a larger ratio p_{k+1}/l_{k+1} has been put after a file f_k with a smaller ratio p_k/l_k.
- For as long as there are inversions there will be inversions of consecutive files and swapping will reduce the expected time. Consequently, the optimal solution is the one with no inversions.

More practice problems for the Greedy Method

- Let X be a set of n intervals on the real line. We say that a set P of points stabs X if every interval in X contains at least one point in P; see the figure below. Describe and analyse an efficient algorithm to compute the smallest set of points that stabs X. Assume that your input consists of two arrays $X_L[1..n]$ and $X_R[1..n]$, representing the left and right endpoints of the intervals in X.

- Is it a good idea to stab the largest possible number of intervals?

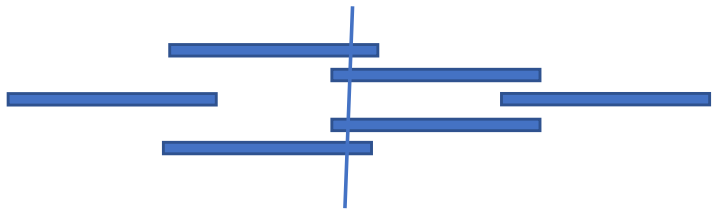


- Hint: the interval which ends the earliest has to be stabbed.
- What is the best place to stab it?

More practice problems for the Greedy Method

- Let X be a set of n intervals on the real line. We say that a set P of points stabs X if every interval in X contains at least one point in P; see the figure below. Describe and analyse an efficient algorithm to compute the smallest set of points that stabs X. Assume that your input consists of two arrays $X_L[1..n]$ and $X_R[1..n]$, representing the left and right endpoints of the intervals in X.

- Is it a good idea to stab the largest possible number of in

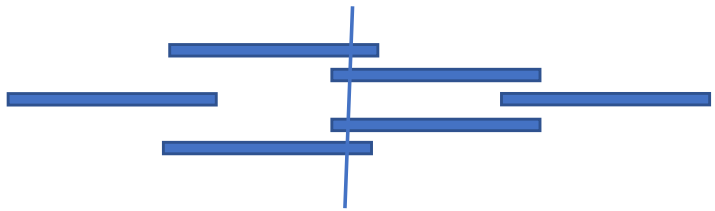


- Hint: the interval which ends the earliest has to be stabbed.
- What is the best place to stab it?

More practice problems for the Greedy Method

- Let X be a set of n intervals on the real line. We say that a set P of points stabs X if every interval in X contains at least one point in P; see the figure below. Describe and analyse an efficient algorithm to compute the smallest set of points that stabs X. Assume that your input consists of two arrays $X_L[1..n]$ and $X_R[1..n]$, representing the left and right endpoints of the intervals in X.

- Is it a good idea to stab the largest possible number of in

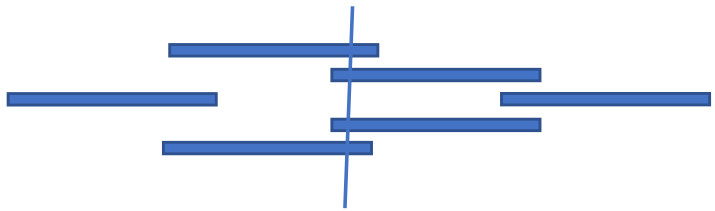


- Hint: the interval which ends the earliest has to be stabbed.
- What is the best place to stab it?

More practice problems for the Greedy Method

- Let X be a set of n intervals on the real line. We say that a set P of points stabs X if every interval in X contains at least one point in P; see the figure below. Describe and analyse an efficient algorithm to compute the smallest set of points that stabs X. Assume that your input consists of two arrays $X_L[1..n]$ and $X_R[1..n]$, representing the left and right endpoints of the intervals in X.

- Is it a good idea to stab the largest possible number of in



- Hint: the interval which ends the earliest has to be stabbed.
- What is the best place to stab it?

0-1 knapsack problem

- **Instance:** A list of weights w_i and values v_i for **discrete items** a_i, $1 \leq i \leq n$, and a maximal weight limit W of your knapsack.
- Tas
  exce
- Can
- Assume there are just three items with weights and val
  (20kg, $100), (30kg, $120) and a knapsack of capacity
- Greedy would choose (10kg, $60) and (20kg, $100), w
  is to take (20kg, $100) and (30kg, $120)!
- So when does the Greedy Strategy work??
- Unfortunately there is no easy rule...

0-1 knapsack problem

- **Instance:** A list of weights w_i and values v_i for **discrete items** a_i, $1 \leq i \leq n$, and a maximal weight limit W of your knapsack.
- **Tas**
  exce
- Can
- Assume there are just three items with weights and val
  (20kg, $100), (30kg, $120) and a knapsack of capacity
- Greedy would choose (10kg, $60) and (20kg, $100), w
  is to take (20kg, $100) and (30kg, $120)!
- So when does the Greedy Strategy work??
- Unfortunately there is no easy rule...

0-1 knapsack problem

- **Instance:** A list of weights w_i and values v_i for **discrete items** a_i, $1 \leq i \leq n$, and a maximal weight limit W of your knapsack.
- **Tas**
  exce
- Can
- Assume there are just three items with weights and val
  (20kg, $100), (30kg, $120) and a knapsack of capacity
- Greedy would choose (10kg, $60) and (20kg, $100), w
  is to take (20kg, $100) and (30kg, $120)!
- So when does the Greedy Strategy work??
- Unfortunately there is no easy rule...

0-1 knapsack problem

- **Instance:** A list of weights w_i and values v_i for **discrete items** a_i, $1 \leq i \leq n$, and a maximal weight limit W of your knapsack.
- **Tas**
  exce
- Can
- Assume there are just three items with weights and val
  (20kg, $100), (30kg, $120) and a knapsack of capacity

- Greedy would choose (10kg, $60) and (20kg, $100), w
  is to take (20kg, $100) and (30kg, $120)!
- So when does the Greedy Strategy work??
- Unfortunately there is no easy rule...

0-1 knapsack problem

- **Instance:** A list of weights w_i and values v_i for **discrete items** a_i, $1 \leq i \leq n$, and a maximal weight limit W of your knapsack.
- **Tas**
  exce
- Can
- Assume there are just three items with weights and val
  (20kg, $100), (30kg, $120) and a knapsack of capacity
-
- Greedy would choose (10kg, $60) and (20kg, $100), w
  is to take (20kg, $100) and (30kg, $120)!
- So when does the Greedy Strategy work??
- Unfortunately there is no easy rule...

0-1 knapsack problem

- **Instance:** A list of weights w_i and values v_i for **discrete items** a_i, $1 \leq i \leq n$, and a maximal weight limit W of your knapsack.
- **Tas**
  exce
- Can
- Assume there are just three items with weights and val
  (20kg, $100), (30kg, $120) and a knapsack of capacity
-
- Greedy would choose (10kg, $60) and (20kg, $100), w
  is to take (20kg, $100) and (30kg, $120)!
- So when does the Greedy Strategy work??
- Unfortunately there is no easy rule...

0-1 knapsack problem

- **Instance:** A list of weights w_i and values v_i for **discrete items** a_i, $1 \leq i \leq n$, and a maximal weight limit W of your knapsack.
- **Tas**
  exce
- Can
- Assume there are just three items with weights and val
  (20kg, $100), (30kg, $120) and a knapsack of capacity
-
- Greedy would choose (10kg, $60) and (20kg, $100), w
  is to take (20kg, $100) and (30kg, $120)!

- So when does the Greedy Strategy work??
- Unfortunately there is no easy rule...

More practice problems for the Greedy Method

- Assume you are given n sorted arrays of different sizes.
- You are allowed to merge any two arrays into a single new sorted array and p
- Des
nu
justification why your algorithm is optimal.
- This problem is somewhat related to the Huffman encoding algorithm, which is, arguably, among the most important applications of the greedy method!

More practice problems for the Greedy Method

- Assume you are given n sorted arrays of different sizes.
- You are allowed to merge any two arrays into a single new sorted array and p
- Des
nu
justification why your algorithm is optimal.
- This problem is somewhat related to the Huffman encoding algorithm, which is, arguably, among the most important applications of the greedy method!

More practice problems for the Greedy Method

- Assume you are given n sorted arrays of different sizes.
- You are allowed to merge any two arrays into a single new sorted array and p
- Des
nu
justification why your algorithm is optimal.
- This problem is somewhat related to the Huffman encoding algorithm, which is, arguably, among the most important applications of the greedy method!

More practice problems for the Greedy Method

- Assume you are given n sorted arrays of different sizes.
- You are allowed to merge any two arrays into a single new sorted array and p
- Des
nu
justification why your algorithm is optimal.
- This problem is somewhat related to the next appli
method, which is, arguably, among the most imp
the greedy method!

The most important applications of the Greedy Method: The Huffman Code

- Assume you are given a set of symbols, for example the English alphabet plus punctuation marks and a blank space (to be used between words).
- Suppose we encode these symbols using binary strings so that sequences of such symbols can be decoded in an unambiguous way.
- One way of doing so is to reserve bit strings of equal and sufficient length, give
plus fo $2^5 = 32$).
- To do bits a
- However this is not an economical way: all the symbols h length but the symbols are not equally frequent.
- One would prefer a coding in which frequent symb 't' have short codes while infrequent ones, such as 'w','x' longer codes.
- However, if the codes are of variable length, then how can we partition a bitstream UNIQUELY into segments each corresponding to a code?
- One way of insuring unique readability of codes from a single bitstream is to ensure that no code of a symbol is a prefix of a code for another symbol.
- Codes with such property are called *the prefix codes*.

The most important applications of the Greedy Method: The Huffman Code

- Assume you are given a set of symbols, for example the English alphabet plus punctuation marks and a blank space (to be used between words).
- You want to encode these symbols using binary strings so that sequences of such symbols can be decoded in an unambiguous way.
- One way of doing so is to reserve bit strings of equal and sufficient length, give
plus 6 ($2^5 = 32$).
- To d
bits a
- However this is not an economical way: all the symbols h
length but the symbols are not equally frequent.
- One would prefer a coding in which frequent symb
't' have short codes while infrequent ones, such as 'w', 'x'
longer codes.
- However, if the codes are of variable length, then how can we partition a bitstream UNIQUELY into segments each corresponding to a code?
- One way of insuring unique readability of codes from a single bitstream is to ensure that no code of a symbol is a prefix of a code for another symbol.
- Codes with such property are called *the prefix codes*.

The most important applications of the Greedy Method: The Huffman Code

- Assume you are given a set of symbols, for example the English alphabet plus punctuation marks and a blank space (to be used between words).
- You want to encode these symbols using binary strings so that sequences of such symbols can be decoded in an unambiguous way.
- One way of doing so is to reserve bit strings of equal and sufficient length, give
plus 6 $2^5 = 32$).
- To d
bits a
- However this is not an economical way: all the symbols h
length but the symbols are not equally frequent.
- One would prefer a coding in which frequent symb
't' have short codes while infrequent ones, such as 'w', 'x'
longer codes.
- However, if the codes are of variable length, then how can we partition a bitstream UNIQUELY into segments each corresponding to a code?
- One way of insuring unique readability of codes from a single bitstream is to ensure that no code of a symbol is a prefix of a code for another symbol.
- Codes with such property are called *the prefix codes*.

The most important applications of the Greedy Method: The Huffman Code

- Assume you are given a set of symbols, for example the English alphabet plus punctuation marks and a blank space (to be used between words).
- You want to encode these symbols using binary strings so that sequences of such symbols can be decoded in an unambiguous way.
- One way of doing so is to reserve bit strings of equal and sufficient length, give
  plus 6 $2^5 = 32$).
- To de
  bits a
- However this is not an economical way: all the symbols h
  length but the symbols are not equally frequent.
- One would prefer a coding in which frequent symb
  't' have short codes while infrequent ones, such as 'w', 'x'
  longer codes.
- However, if the codes are of variable length, then how can we partition a bitstream UNIQUELY into segments each corresponding to a code?
- One way of insuring unique readability of codes from a single bitstream is to ensure that no code of a symbol is a prefix of a code for another symbol.
- Codes with such property are called *the prefix codes*.

The most important applications of the Greedy Method: The Huffman Code

- Assume you are given a set of symbols, for example the English alphabet plus punctuation marks and a blank space (to be used between words).
- You want to encode these symbols using binary strings so that sequences of such symbols can be decoded in an unambiguous way.
- One way of doing so is to reserve bit strings of equal and sufficient length, give
  plus 6 $2^5 = 32$).
- To de
  bits a
- However this is not an economical way: all the symbols h
  length but the symbols are not equally frequent.
- One would prefer an encoding in which frequent symb
  't' have short codes while infrequent ones, such as 'w', 'x'
  longer codes.
- However, if the codes are of variable length, then how can we partition a bitstream UNIQUELY into segments each corresponding to a code?
- One way of insuring unique readability of codes from a single bitstream is to ensure that no code of a symbol is a prefix of a code for another symbol.
- Codes with such property are called *the prefix codes*.

The most important applications of the Greedy Method: The Huffman Code

- Assume you are given a set of symbols, for example the English alphabet plus punctuation marks and a blank space (to be used between words).
- You want to encode these symbols using binary strings so that sequences of such symbols can be decoded in an unambiguous way.
- One way of doing so is to reserve bit strings of equal and sufficient length, give
  plus 6 $2^5 = 32$).
- To de
  bits a
- However this is not an economical way: all the symbols h
  length but the symbols are not equally frequent.
- One would prefer an encoding in which frequent symb
  't' have short codes while infrequent ones, such as 'w','x
  longer codes.
- However, if the codes are of variable length, then how can we partition a bitstream UNIQUELY into segments each corresponding to a code?
- One way of insuring unique readability of codes from a single bitstream is to ensure that no code of a symbol is a prefix of a code for another symbol.
- Codes with such property are called *the prefix codes*.

The most important applications of the Greedy Method: The Huffman Code

- Assume you are given a set of symbols, for example the English alphabet plus punctuation marks and a blank space (to be used between words).
- You want to encode these symbols using binary strings so that sequences of such symbols can be decoded in an unambiguous way.
- One way of doing so is to reserve bit strings of equal and sufficient length, give
  plus 6 $2^5 = 32$).
- To de
  bits a
- However this is not an economical way: all the symbols h
  length but the symbols are not equally frequent.
- One would prefer an encoding in which frequent symb
  't' have short codes while infrequent ones, such as 'w','x
  longer codes.
- However, if the codes are of variable length, then how can we partition a bitstream UNIQUELY into segments each corresponding to a code?
- One way of insuring unique readability of codes from a single bitstream is to ensure that no code of a symbol is a prefix of a code for another symbol.
- Codes with such property are called *the prefix codes*.

The most important applications of the Greedy Method: The Huffman Code

- Assume you are given a set of symbols, for example the English alphabet plus punctuation marks and a blank space (to be used between words).
- You want to encode these symbols using binary strings so that sequences of such symbols can be decoded in an unambiguous way.
- One way of doing so is to reserve bit strings of equal and sufficient length, give
  plus 6 $2^5 = 32$).
- To de
  bits a
- However this is not an economical way: all the symbols h
  length but the symbols are not equally frequent.
- One would prefer an encoding in which frequent symb
  't' have short codes while infrequent ones, such as 'w','x
  longer codes.
- However, if the codes are of variable length, then how can we partition a bitstream UNIQUELY into segments each corresponding to a code?
- One way of insuring unique readability of codes from a single bitstream is to ensure that no code of a symbol is a prefix of a code for another symbol.
- Codes with such property are called *the prefix codes*.

The most important applications of the Greedy Method: The Huffman Code

- Assume you are given a set of symbols, for example the English alphabet plus punctuation marks and a blank space (to be used between words).
- You want to encode these symbols using binary strings so that sequences of such symbols can be decoded in an unambiguous way.
- One way of doing so is to reserve bit strings of equal and sufficient length, give
  plus 6 $2^5 = 32$).
- To de
  bits a
- However this is not an economical way: all the symbols h
  length but the symbols are not equally frequent.
- One would prefer an encoding in which frequent symb
  't' have short codes while infrequent ones, such as 'w','x
  longer codes.
- However, if the codes are of variable length, then how can we partition a bitstream UNIQUELY into segments each corresponding to a code?
- One way of insuring unique readability of codes from a single bitstream is to ensure that no code of a symbol is a prefix of a code for another symbol.
- Codes with such property are called *the prefix codes*.

The most important applications of the Greedy Method: The Huffman Code

- We can now formulate the problem:

  Give
  opti
  enc

- Note that this amounts to saying that the l
  in an "average" text is as small as possible.

- We now sketch the algorithm informally; please see th
  and the proof of optimality.

Greedy Method applied to graphs

- There are n radio towers for broadcasting tsunami warnings. You are given the (x, y) coordinates of each tower and its radius of range. When a tow
will a

- You
whe
towers will eventually get activated and send a tsu

- The goal is to design an algorithm which finds the few towers you must equip with seismic sensors.

Greedy Method applied to graphs

- There are n radio towers for broadcasting tsunami warnings. You are given the (x, y) coordinates of each tower and its radius of range. When a tow
will a
- You
whe
towers will eventually get activated and send a tsu
- The goal is to design an algorithm which finds the few towers you must equip with seismic sensors.

Greedy Method applied to graphs

- There are n radio towers for broadcasting tsunami warnings. You are given the (x, y) coordinates of each tower and its radius of range. When a tow
will a
- You
whe
towers will eventually get activated and send a tsu
- The goal is to design an algorithm which finds the few towers you must equip with seismic sensors.

Greedy Method applied to graphs

- Someone has proposed the following two algorithms:
  1. Algorithm 1: Find the unactivated tower with the largest radius (if multiple with the same radius, pick the any). Activate this tower. Find and remove all activated towers. Repeat.
  2. Algorithm 2: Find the unactivated tower with the smallest radius (if multiple with the same radius, pick the any). Activate this tower. Find and remove all activated towers. Repeat.
- Give examples which show that neither Algorithm 1 nor Algorithm 2 solve the problem correctly.
- Design an algorithm which correctly solves the problem.
- Solving this problem involves several important concepts which we now revisit.

Greedy Method applied to graphs

- Someone has proposed the following two algorithms:
  1. Algorithm 1: Find the unactivated tower with the largest radius (if multiple with the same radius, pick the any of them). Activate this tower. Find and remove all activated towers. Repeat.
  2.
- Give examples which show that neither Algorithm solves the problem correctly.
- Design an algorithm which correctly solves the problem.
- Solving this problem involves several important concepts which we now revisit.

Greedy Method applied to graphs

- Someone has proposed the following two algorithms:
  1. Algorithm 1: Find the unactivated tower with the largest radius (if multiple with the same radius, pick the any of them). Activate this tower. Find and remove all activated towers. Repeat.
  2.

- Give examples which show that neither Algorit solve the problem correctly.
- Design an algorithm which correctly solves the problem.
- Solving this problem involves several important concepts which we now revisit.

Greedy Method applied to graphs

- Someone has proposed the following two algorithms:
  1. Algorithm 1: Find the unactivated tower with the largest radius (if multiple with the same radius, pick the any of them). Activate this tower. Find and remove all activated towers. Repeat.
  2. [illegible]
- Give examples which show that neither Algorit solve the problem correctly.
- Design an algorithm which correctly solves the problem.
- Solving this problem involves several important concepts which we now revisit.

Greedy Method applied to graphs

- Someone has proposed the following two algorithms:
  1. Algorithm 1: Find the unactivated tower with the largest radius (if multiple with the same radius, pick the any of them). Activate this tower. Find and remove all activated towers. Repeat.
  2.
- Give examples which show that neither Algorit solve the problem correctly.
- Design an algorithm which correctly solves the problem.
- Solving this problem involves several important concepts which we now revisit.

Greedy Method applied to graphs

- Someone has proposed the following two algorithms:
  1. Algorithm 1: Find the unactivated tower with the largest radius (if multiple with the same radius, pick the any of them). Activate this tower. Find and remove all activated towers. Repeat.
  2. [illegible]
- Give examples which show that neither Algorit solve the problem correctly.
- Design an algorithm which correctly solves the problem.
- Solving this problem involves several important concepts which we now revisit.

Greedy Method applied to graphs

- Given a directed graph $G = (V, E)$ and a vertex v, the *strongly connected component* of G containing v consists of all vertices $u \in V$ such that there is a path in G from v to u and a path from u to v.

- How do we find a strongly connected component $C \subseteq V$ containing u?

- Con
vert
dire
of
he

- Use BFS to find the set $D \subseteq V$ of all vertices in V which are reachable in G from v.

- Find also the set $R \subseteq V$ of all vertices which can reach v in G from v.

- The strongly connected component C of G containing v is just the set $C = D \cap R$.

- Clearly, distinct strongly connected components are disjoint sets.

Greedy Method applied to graphs

- Given a directed graph $G = (V, E)$ and a vertex v, the *strongly connected component* of G containing v consists of all vertices $u \in V$ such that there is a path in G from v to u and a path from u to v.
- How do we find a strongly connected component $C \subseteq V$ containing u?
- Con
vert
dire
- Use BFS to find the set $D \subseteq V$ of all vertices in V which are reachable in G from v.
- Find also the set $R \subseteq V$ of all vertices which
v.
- The strongly connected component C of G containing v is just the set $C = D \cap R$.
- Clearly, distinct strongly connected components are disjoint sets.

Greedy Method applied to graphs

- Given a directed graph $G = (V, E)$ and a vertex v, the *strongly connected component* of G containing v consists of all vertices $u \in V$ such that there is a path in G from v to u and a path from u to v.
- How do we find a strongly connected component $C \subseteq V$ containing u?
- Con
vert
dire
of
he
- Use BFS to find the set $D \subseteq V$ of all vertices in V which are reachable in G from v.
- Find also the set $R \subseteq V$ of all vertices which can reach v in G from v.
- The strongly connected component C of G containing v is just the set $C = D \cap R$.
- Clearly, distinct strongly connected components are disjoint sets.

Greedy Method applied to graphs

- Given a directed graph $G = (V, E)$ and a vertex v, the *strongly connected component* of G containing v consists of all vertices $u \in V$ such that there is a path in G from v to u and a path from u to v.
- How do we find a strongly connected component $C \subseteq V$ containing u?
- Con
  vert
  dire
  of
  he
- Use BFS to find the set $D \subseteq V$ of all vertices in V which are reachable in G from v.
- Find also the set $R \subseteq V$ of all vertices which can reach v in G from v.
- The strongly connected component C of G containing v is just the set $C = D \cap R$.
- Clearly, distinct strongly connected components are disjoint sets.

Greedy Method applied to graphs

- Given a directed graph $G = (V, E)$ and a vertex v, the *strongly connected component* of G containing v consists of all vertices $u \in V$ such that there is a path in G from v to u and a path from u to v.
- How do we find a strongly connected component $C \subseteq V$ containing u?
- Con
  vert
  dire
  of
  he
- Use BFS to find the set $D \subseteq V$ of all vertices in V which are reachable in G from v.
- Find also the set $R \subseteq V$ of all vertices whic
  v from
  v.
- The strongly connected component C of G containing v is just the set $C = D \cap R$.
- Clearly, distinct strongly connected components are disjoint sets.

Greedy Method applied to graphs

- Given a directed graph $G = (V, E)$ and a vertex v, the *strongly connected component* of G containing v consists of all vertices $u \in V$ such that there is a path in G from v to u and a path from u to v.
- How do we find a strongly connected component $C \subseteq V$ containing u?
- Con
  vert
  dire
  of
  he
- Use BFS to find the set $D \subseteq V$ of all vertices in V which are reachable in G from v.
- Find also the set $R \subseteq V$ of all vertices whic
  v from
  v.
- The strongly connected component C of G containing v is just the set $C = D \cap R$.
- Clearly, distinct strongly connected components are disjoint sets.

Greedy Method applied to graphs

- Given a directed graph $G = (V, E)$ and a vertex v, the *strongly connected component* of G containing v consists of all vertices $u \in V$ such that there is a path in G from v to u and a path from u to v.
- How do we find a strongly connected component $C \subseteq V$ containing u?
- Con
  vert
  dire
  of
  he
- Use BFS to find the set $D \subseteq V$ of all vertices in V which are reachable in G from v.
- Find also the set $R \subseteq V$ of all vertices whic
  v from
  v.
- The strongly connected component C of G containing v is just the set $C = D \cap R$.
- Clearly, distinct strongly connected components are disjoint sets.

Greedy Method applied to graphs

- Let S_G be the set of all strongly connected components of a graph G.
- We define a graph $\Gamma = (S_G, E^*)$ with vertices S_G and directed edges E^* between the strongly connected components so that if $\sigma_1 \in S_G$ and σ_2 / $_2$ in E^* just in cas at there exis
- Cle $_G$ topological sorting of vertices.
- Topological sort of a directed acyclic graph is a linea (enumeration) of its vertices $\sigma_i \in S_G$ s $(\sigma_i, \sigma_j) \in E^*$ then vertex σ_i precedes σ_j in such an ordering , i.e., $i < j$.
- How do we topologically sort a directed acyclic graphs?

Greedy Method applied to graphs

- Let S_G be the set of all strongly connected components of a graph G.
- We define a graph $\Sigma = (S_G, E^*)$ with vertices S_G and directed edges E^* between the strongly connected components so that if $\sigma_1 \in S_G$ and σ_2 in cas / $_2$ in E^* just exis at there
- Cle $_G$ topological sorting of vertices.
- Topological sort of a directed acyclic graph is a linear (enumeration) of its vertices $\sigma_i \in S_G$ s $(\sigma_i, \sigma_j) \in E^*$ then vertex σ_i precedes σ_j in such an ordering , i.e., $i < j$.
- How do we topologically sort a directed acyclic graphs?

Greedy Method applied to graphs

- Let S_G be the set of all strongly connected components of a graph G.
- We define a graph $\Sigma = (S_G, E^*)$ with vertices S_G and directed edges E^* between the strongly connected components so that if $\sigma_1 \in S_G$ and σ_2 in E^* just in cas at there exis
- Cle topological sorting of vertices.
- Topological sort of a directed acyclic graph is a linear (enumeration) of its vertices $\sigma_i \in S_G$ s $(\sigma_i, \sigma_j) \in E^*$ then vertex σ_i precedes σ_j in such an ordering , i.e., $i < j$.
- How do we topologically sort a directed acyclic graphs?

Greedy Method applied to graphs

- Let S_G be the set of all strongly connected components of a graph G.
- We define a graph $\Sigma = (S_G, E^*)$ with vertices S_G and directed edges E^* between the strongly connected components so that if $\sigma_1 \in S_G$ and σ_2 in cas exis $_2$ in E^* just at there
- Cle topological sorting of vertices.
- Topological sort of a directed acyclic graph is a linea (enumeration) of its vertices $\sigma_i \in S_G$ s $(\sigma_i, \sigma_j) \in E^*$ then vertex σ_i precedes σ_j in such an ordering , i.e., $i < j$.
- How do we topologically sort a directed acyclic graphs?

Greedy Method applied to graphs

- Let S_G be the set of all strongly connected components of a graph G.
- We define a graph $\Sigma = (S_G, E^*)$ with vertices S_G and directed edges E^* between the strongly connected components so that if $\sigma_1 \in S_G$ and σ_2 in E^* just in cas at there exis
- Cle topological sorting of vertices.
- Topological sort of a directed acyclic graph is a linea (enumeration) of its vertices $\sigma_i \in S_G$ s $(\sigma_i, \sigma_j) \in E^*$ then vertex σ_i precedes σ_j in such an ordering , i.e., $i < j$.
- How do we topologically sort a directed acyclic graphs?

Topological sorting

- $L \leftarrow$ Empty list that will contain ordered elements
- $S \leftarrow$ Set of all nodes with no incoming edge
- **while** S is non-empty **do**
  - remove a node u from S;
  - add u to L;
  - for each node v with an edge e from u to v do
    - remove edge e from the graph;
    - if v has no other incoming edges
- **if** graph has edges left, **then return** e

else return L (a topologically sorted order)

Now it should be easy to use the Greedy Strategy to solve the problem of finding the fewest number of towers which you must equip with seismic sensors, so that all emergency transmission towers get activated.

Topological sorting

- $L \leftarrow$ Empty list that will contain ordered elements
- $S \leftarrow$ Set of all nodes with no incoming edge

- **while** S is not empty **do**
  - remove a node u from S;
  -
  - remove edge e from the graph;
  - has no other incoming edges
- **if** graph has edges left, **then return** e

else return L (a topologically sorted order)

Now it should be easy to use the Greedy Strategy to solve the problem of finding the fewest number of towers which you must equip with seismic sensors, so that all emergency transmission towers get activated.

Topological sorting

- $L \leftarrow$ Empty list that will contain ordered elements
- $S \leftarrow$ Set of all nodes with no incoming edge
- **while** S is non-empty **do**
  - remove a node u from S;
  - remove edge e from the graph;
  - if u has no other incoming edges
- **if** graph has edges left, **then return** e

else return L (a topologically sorted order)

Now it should be easy to use the Greedy Strategy to solve the problem of finding the fewest number of towers which you must equip with seismic sensors, so that all emergency transmission towers get activated.

Topological sorting

- $L \leftarrow$ Empty list that will contain ordered elements
- $S \leftarrow$ Set of all nodes with no incoming edge
- **while** S is non-empty **do**
  - remove a node u from S;
  - remove edge e from the graph;
  - if u has no other incoming edges
- **if** graph has edges left, **then return** e

else return L (a topologically sorted order)

Now it should be easy to use the Greedy Strategy to solve the problem of finding the fewest number of towers which you must equip with seismic sensors, so that all emergency transmission towers get activated.

Topological sorting

- $L \leftarrow$ Empty list that will contain ordered elements
- $S \leftarrow$ Set of all nodes with no incoming edge
- **while** S is non-empty **do**
  - remove a node u from S;
  - remove edge e from the graph;
  - if u has no other incoming edges
- **if** graph has edges left, **then return** e

else return L (a topologically sorted order)

Now it should be easy to use the Greedy Strategy to solve the problem of finding the fewest number of towers which you must equip with seismic sensors, so that all emergency transmission towers get activated.

Topological sorting

- $L \leftarrow$ Empty list that will contain ordered elements
- $S \leftarrow$ Set of all nodes with no incoming edge
- **while** S is non-empty **do**
  - remove a node u from S;
  - remove edge e from the graph;
  - **if** v has no other incoming edges
- **if** graph has edges left, **then return** e

  else return L (a topologically sorted order)

Now it should be easy to use the Greedy Strategy to solve the problem of finding the fewest number of towers which you must equip with seismic sensors, so that all emergency transmission towers get activated.

Topological sorting

- $L \leftarrow$ Empty list that will contain ordered elements
- $S \leftarrow$ Set of all nodes with no incoming edge
- **while** S is non-empty **do**
  - remove a node u from S;
  - remove edge e from the graph;
  - **if** v has no other incoming edges
- **if** graph has edges left, **then return** e

else return L (a topologically sorted order)

Now it should be easy to use the Greedy Strategy to solve the problem of finding the fewest number of towers which you must equip with seismic sensors, so that all emergency transmission towers get activated.

Topological sorting

- $L \leftarrow$ Empty list that will contain ordered elements
- $S \leftarrow$ Set of all nodes with no incoming edge
- **while** S is non-empty **do**
  - remove a node u from S;
  - remove edge e from the graph;
  - **if** v has no other incoming edges
- **if** graph has edges left, **then return** e

  else return L (a topologically sorted order)

Now it should be easy to use the Greedy Strategy to solve the problem of finding the fewest number of towers which you must equip with seismic sensors, so that all emergency transmission towers get activated.

Topological sorting

- $L \leftarrow$ Empty list that will contain ordered elements
- $S \leftarrow$ Set of all nodes with no incoming edge
- **while** S is non-empty **do**
  - remove a node u from S;
  - remove edge e from the graph;
  - **if** v has no other incoming edges
- **if** graph has edges left, **then return** e

  else return L (a topologically sorted order)

Now it should be easy to use the Greedy Strategy to solve the problem of finding the fewest number of towers which you must equip with seismic sensors, so that all emergency transmission towers get activated.

An augmented Priority Queue

- We will need a priority queue which allows an efficient change of the key of an element in the queue, so we first need to extend the Heap data structure.

- We will use heaps represented by arrays: the left child of $A[i]$ is stored in $A[2i]$ and the right child in $A[2i+1]$.

- We
way
to n

- Thus, every element of A is of the form $(i, k(i))$ where $k(i)$ is the key of element i.

- Besides the array A which represents the
P of the same length which stores the position of ele
thus $A[P[i]] = (i, k(i))$.

- Changing the key of an element i is now an $O(\log n)$ operation: we look up its position $P[i]$ in A, change the key of the element in $A[P[i]]$ and then perform the Heappify operation to make sure the Heap property is being preserved.

An augmented Priority Queue

- We will need a priority queue which allows an efficient change of the key of an element in the queue, so we first need to extend the Heap data structure.
- We will use heaps represented by arrays; the left child of $A[i]$ is stored in $A[2i]$ and the right child in $A[2i+1]$.
- We way to n
- Thus, every element of A is of the form $(i, k(i))$ where $k(i)$ is the key of element i.
- Besides the array A which represents the P of the same length which stores the position of ele thus $A[P[i]] = (i, k(i))$.
- Changing the key of an element i is now an $O(\log n)$ operation: we look up its position $P[i]$ in A, change the key of the element in $A[P[i]]$ and then perform the Heappify operation to make sure the Heap property is being preserved.

An augmented Priority Queue

- We will need a priority queue which allows an efficient change of the key of an element in the queue, so we first need to extend the Heap data structure.
- We will use heaps represented by arrays; the left child of $A[i]$ is stored in $A[2i]$ and the right child in $A[2i+1]$.
- We
way
to n s 1
- Thus, every element of A is of the form $(i, k(i))$ where $k(i)$ is the key of element i.
- Besides the array A which represents the
P of the same length which stores the position of ele
thus $A[P[i]] = (i, k(i))$.
- Changing the key of an element i is now an $O(\log n)$ operation: we look up its position $P[i]$ in A, change the key of the element in $A[P[i]]$ and then perform the Heappify operation to make sure the Heap property is being preserved.

An augmented Priority Queue

- We will need a priority queue which allows an efficient change of the key of an element in the queue, so we first need to extend the Heap data structure.
- We will use heaps represented by arrays; the left child of $A[i]$ is stored in $A[2i]$ and the right child in $A[2i+1]$.
- We way to n s 1
- Thus, every element of A is of the form $(i, k(i))$ where $k(i)$ is the key of element i.
- Besides the array A which represents the P of the same length which stores the position of ele thus $A[P[i]] = (i, k(i))$.
- Changing the key of an element i is now an $O(\log n)$ operation: we look up its position $P[i]$ in A, change the key of the element in $A[P[i]]$ and then perform the Heappify operation to make sure the Heap property is being preserved.

An augmented Priority Queue

- We will need a priority queue which allows an efficient change of the key of an element in the queue, so we first need to extend the Heap data structure.
- We will use heaps represented by arrays; the left child of $A[i]$ is stored in $A[2i]$ and the right child in $A[2i+1]$.
- We
way
to n s 1
- Thus, every element of A is of the form $(i, k(i))$ where $k(i)$ is the key of element i.
- Besides the array A which represents the h
P of the same length which stores the position of ele
thus $A[P[i]] = (i, k(i))$.
- Changing the key of an element i is now an $O(\log n)$ operation: we look up its position $P[i]$ in A, change the key of the element in $A[P[i]]$ and then perform the Heappify operation to make sure the Heap property is being preserved.

An augmented Priority Queue

- We will need a priority queue which allows an efficient change of the key of an element in the queue, so we first need to extend the Heap data structure.
- We will use heaps represented by arrays; the left child of $A[i]$ is stored in $A[2i]$ and the right child in $A[2i+1]$.
- We
way
to n s 1
- Thus, every element of A is of the form $(i, k(i))$ where $k(i)$ is the key of element i.
- Besides the array A which represents the h
P of the same length which stores the position of ele
thus $A[P[i]] = (i, k(i))$.
- Changing the key of an element i is now an $O(\log n)$ operation: we look up its position $P[i]$ in A, change the key of the element in $A[P[i]]$ and then perform the Heappify operation to make sure the Heap property is being preserved.

An augmented Priority Queue

Heap Array

i_6	i_4	i_5	i_2	i_3	i_1	i_7	i_8
1	2	6	3	9	7	11	10
1	2	3	4	5	6	7	8

Index Array

i_1	i_2	i_3	i_4	i_5	i_6	i_7	i_8
6	4	5	2	3	1	7	8

Greedy Method applied to graphs: Shortest Paths

- Some of the most important applications of the Greedy Strategy are in graph algorithms.
- Assume we are given a directed graph $G = (V, E)$ with **non-negative** wei
- We a
- For simplicity, we will assume that for every v to u.
- The task is to find for every $u \in V$ the s
- This is accomplished by a very elegant greedy algorithm by Edsger Dijkstra already in 1959!

Greedy Method applied to graphs: Shortest Paths

- Some of the most important applications of the Greedy Strategy are in graph algorithms.
- Assume we are given a directed graph $G = (V, E)$ with **non-negative** wei
- We a
- For simplicity, we will assume that for every v to u.
- The task is to find for every $u \in V$ the s
- This is accomplished by a very elegant greedy algorithm by Edsger Dijkstra already in 1959!

Greedy Method applied to graphs: Shortest Paths

- Some of the most important applications of the Greedy Strategy are in graph algorithms.
- Assume we are given a directed graph $G = (V, E)$ with **non-negative** wei
- We a
- For simplicity, we will assume that for every v to u.
- The task is to find for every $u \in V$ the s
- This is accomplished by a very elegant greedy algorithm by Edsger Dijkstra already in 1959!

Greedy Method applied to graphs: Shortest Paths

- Some of the most important applications of the Greedy Strategy are in graph algorithms.
- Assume we are given a directed graph $G = (V, E)$ with **non-negative** wei
- We a
- For simplicity, we will assume that for every v to u.
- The task is to find for every $u \in V$ the s
- This is accomplished by a very elegant greedy algorithm by Edsger Dijkstra already in 1959!

Greedy Method applied to graphs: Shortest Paths

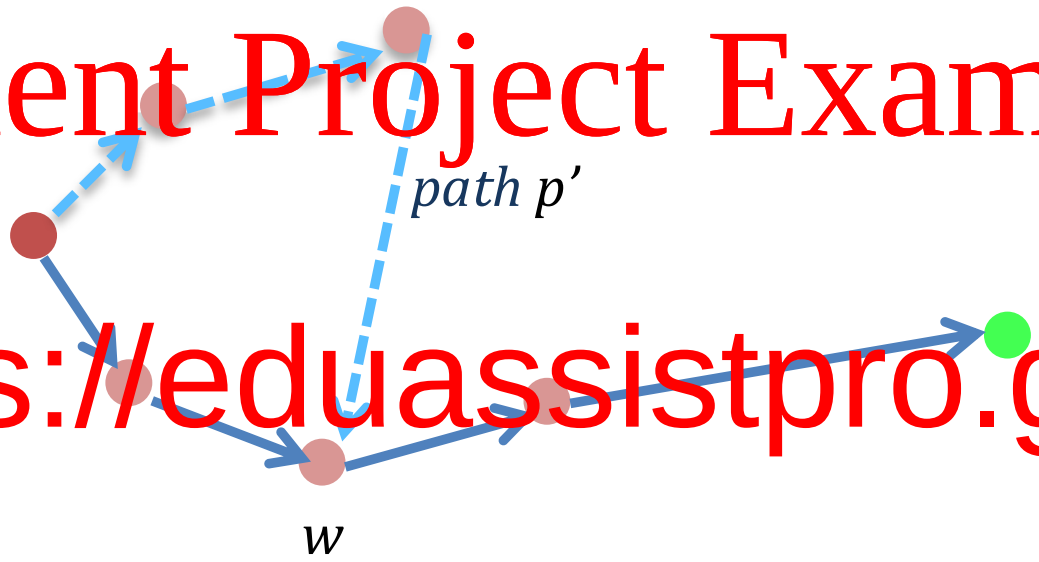
- Some of the most important applications of the Greedy Strategy are in graph algorithms.
- Assume we are given a directed graph $G = (V, E)$ with **non-negative** wei
- We a
- For simplicity, we will assume that for every v to u.
- The task is to find for every $u \in V$ the s .
- This is accomplished by a very elegant greedy algorithm by Edsger Dijkstra already in 1959!

Greedy Method applied to graphs: Shortest Paths

- Some of the most important applications of the Greedy Strategy are in graph algorithms.
- Assume we are given a directed graph $G = (V, E)$ with **non-negative** wei
- We a
- For simplicity, we will assume that for every v to u.
- The task is to find for every $u \in V$ the s .
- This is accomplished by a very elegant greedy algorithm by Edsger Dijkstra already in 1959!

Greedy Method applied to graphs: Shortest Paths

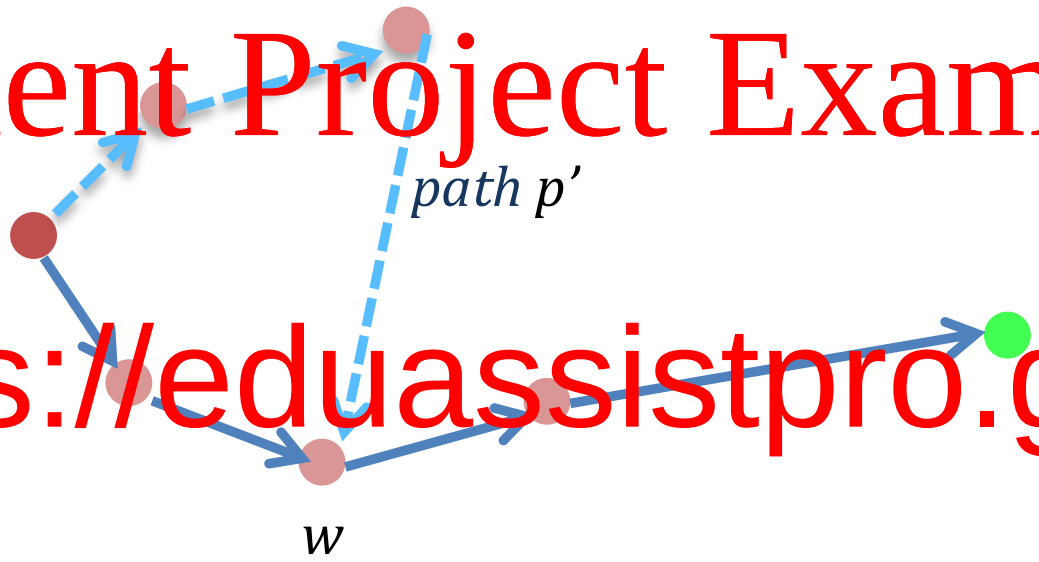
- We first prove a simple fact about shortest paths.
- Consider a shortest path p in G from a vertex v to a vertex z (shown in dark blue).



- We claim that for every vertex w on the path p, the path from v to w is just the truncation of the shortest path fro at w.
- Assume the opposite, that there is a shorter path fro own in dashed light blue) which is not an initial segment of the shortest path from v to z.
- However, in that case we could remove the part of the shortest path between v and z which is between v and w and replaced it with the light blue shorter path from v to w, thus obtaining a shorter path from v to z.
- Contradiction!

Greedy Method applied to graphs: Shortest Paths

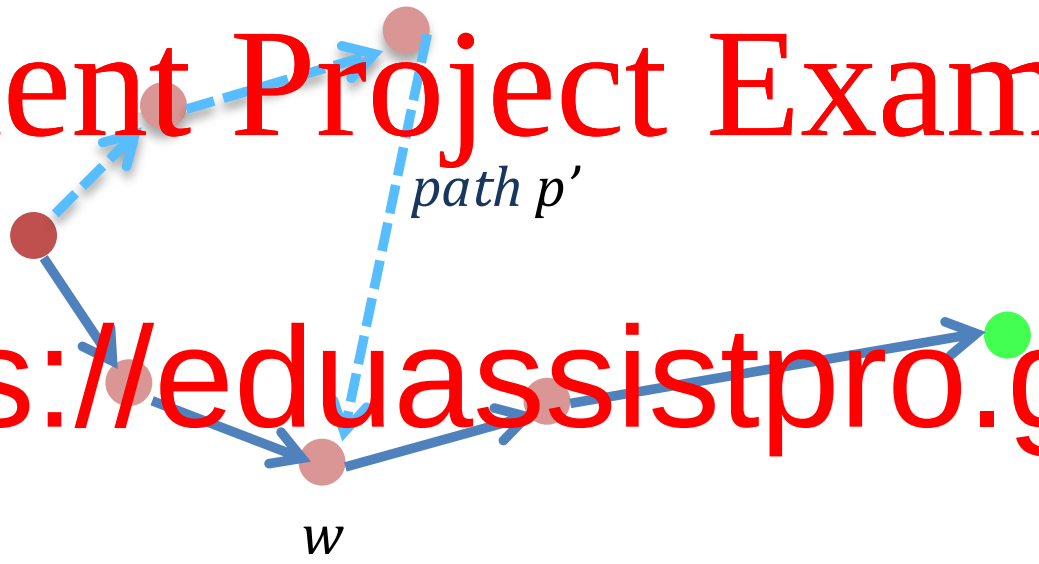
- We first prove a simple fact about shortest paths.
- Consider a shortest path p in G from a vertex v to a vertex z (shown in dark blue).



- We claim that for every vertex w on the path p, the path from v to w is just the truncation of the shortest path fro at v.
- Assume the opposite, that there is a shorter path fro own in dashed light blue) which is not an initial segment of the shortest path from v to z.
- However, in that case we could remove the part of the shortest path between v and z which is between v and w and replaced it with the light blue shorter path from v to w, thus obtaining a shorter path from v to z.
- Contradiction!

Greedy Method applied to graphs: Shortest Paths

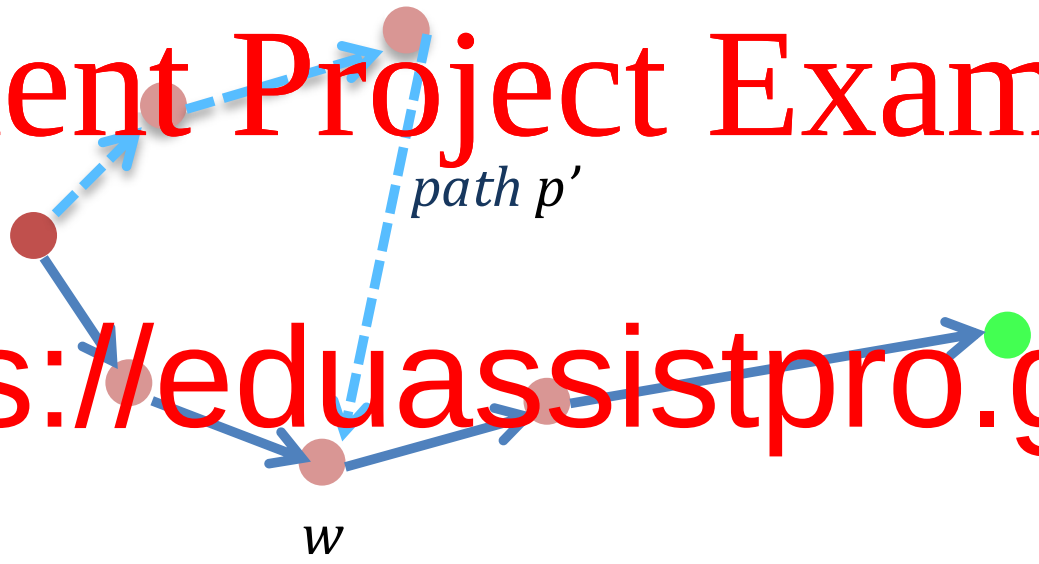
- We first prove a simple fact about shortest paths.
- Consider a shortest path p in G from a vertex v to a vertex z (shown in dark blue).



- We claim that for every vertex w on that p to w is just the truncation of the shortest path fro at w.
- Assume the opposite, that there is a shorter path fro own in dashed light blue) which is not an initial segment of the shortest path from v to z.
- However, in that case we could remove the part of the shortest path between v and z which is between v and w and replaced it with the light blue shorter path from v to w, thus obtaining a shorter path from v to z.
- Contradiction!

Greedy Method applied to graphs: Shortest Paths

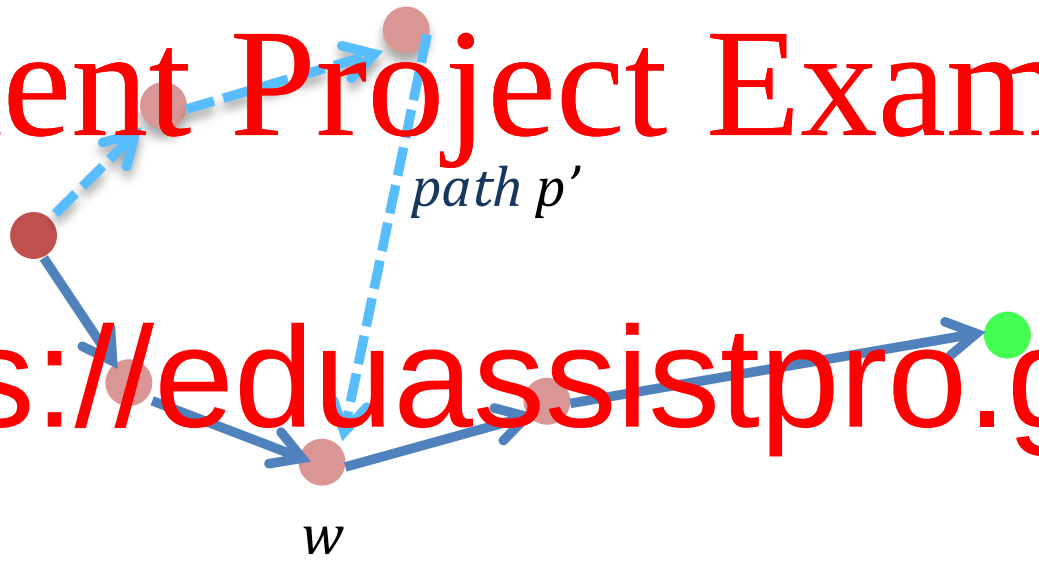
- We first prove a simple fact about shortest paths.
- Consider a shortest path p in G from a vertex v to a vertex z (shown in dark blue).



- We claim that for every vertex w on that p to w is just the truncation of the shortest path fro at w.
- Assume the opposite, that there is a shorter path fro own in dashed light blue) which is not an initial segment of the shortest path from v to z.
- However, in that case we could remove the part of the shortest path between v and z which is between v and w and replaced it with the light blue shorter path from v to w, thus obtaining a shorter path from v to z.
- Contradiction!

Greedy Method applied to graphs: Shortest Paths

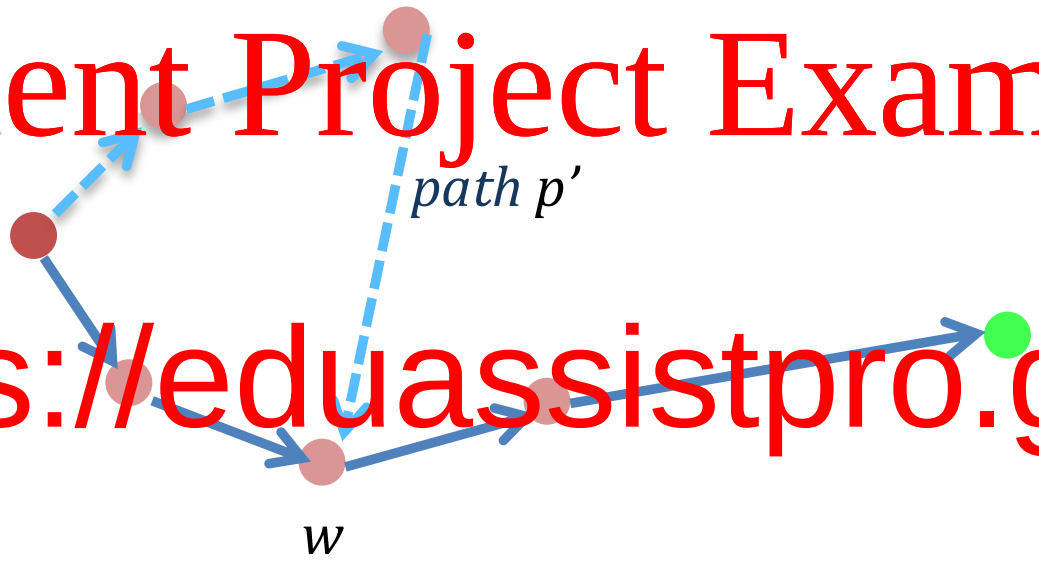
- We first prove a simple fact about shortest paths.
- Consider a shortest path p in G from a vertex v to a vertex z (shown in dark blue).



- We claim that for every vertex w on that v to w is just the truncation of the shortest path fro at w.
- Assume the opposite, that there is a shorter path fro own in dashed light blue) which is not an initial segment of the shortest path from v to z.
- However, in that case we could remove the part of the shortest path between v and z which is between v and w and replaced it with the light blue shorter path from v to w, thus obtaining a shorter path from v to z.
- Contradiction!

Greedy Method applied to graphs: Shortest Paths

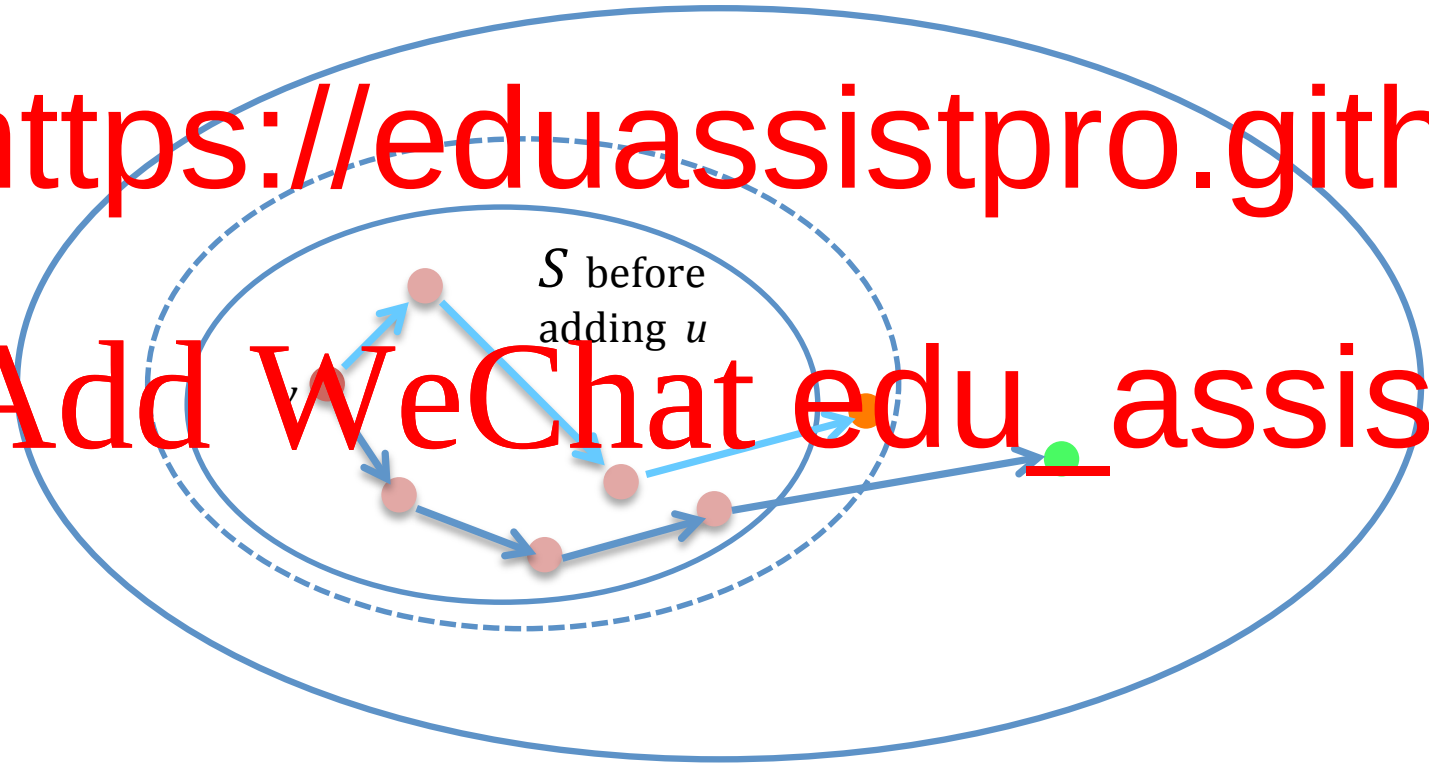
- We first prove a simple fact about shortest paths.
- Consider a shortest path p in G from a vertex v to a vertex z (shown in dark blue).



- We claim that for every vertex w on that v to w is just the truncation of the shortest path fro at w.
- Assume the opposite, that there is a shorter path fro own in dashed light blue) which is not an initial segment of the shortest path from v to z.
- However, in that case we could remove the part of the shortest path between v and z which is between v and w and replaced it with the light blue shorter path from v to w, thus obtaining a shorter path from v to z.
- Contradiction!

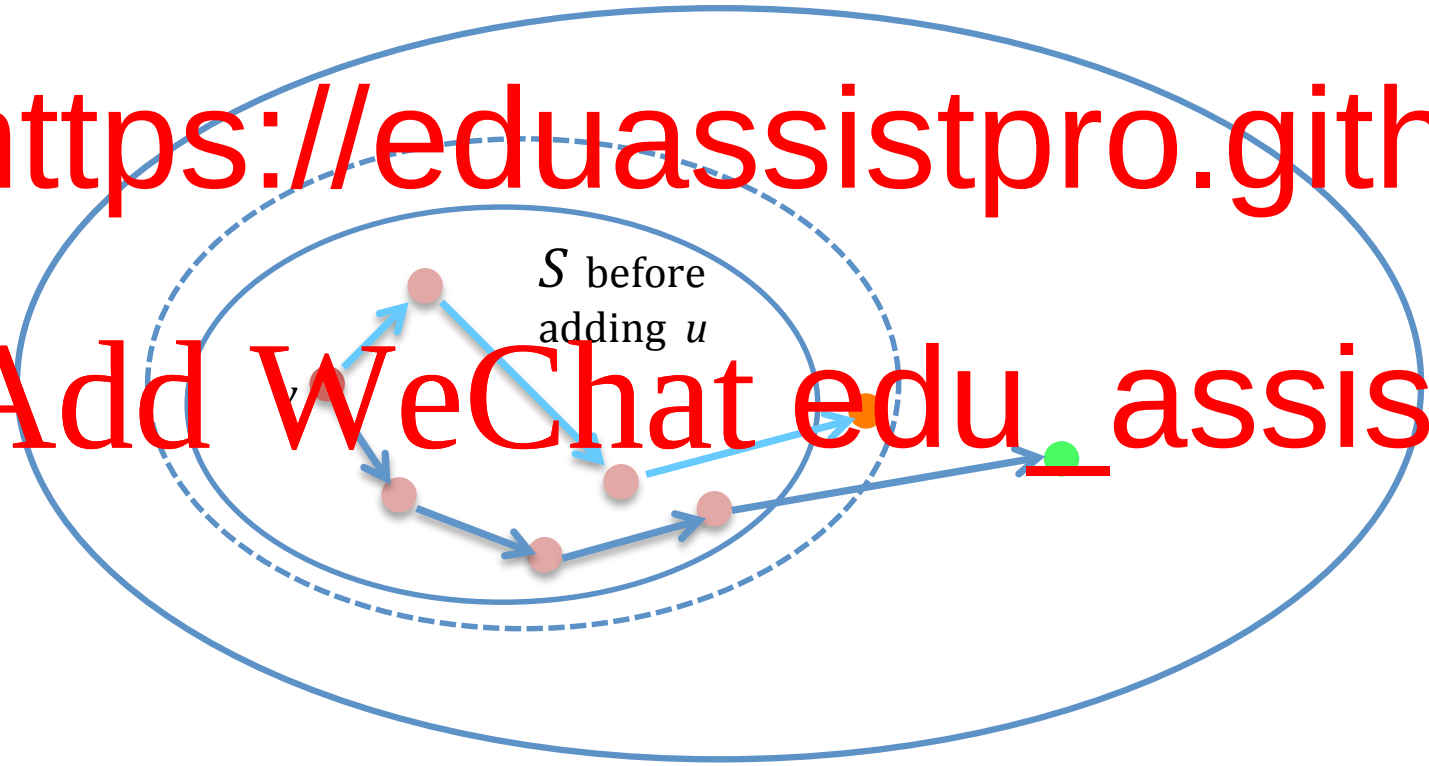
Dijkstra's Shortest Paths Algorithm

- The algorithm builds a set S of vertices for which the shortest path has been already established, starting with a single source vertex $S = \{v\}$ and adding one vertex at a time.
- At each stage of the construction we add the element $u \in V \setminus S$ which has the shortest path from v to u with all intermediate vertices already in S.



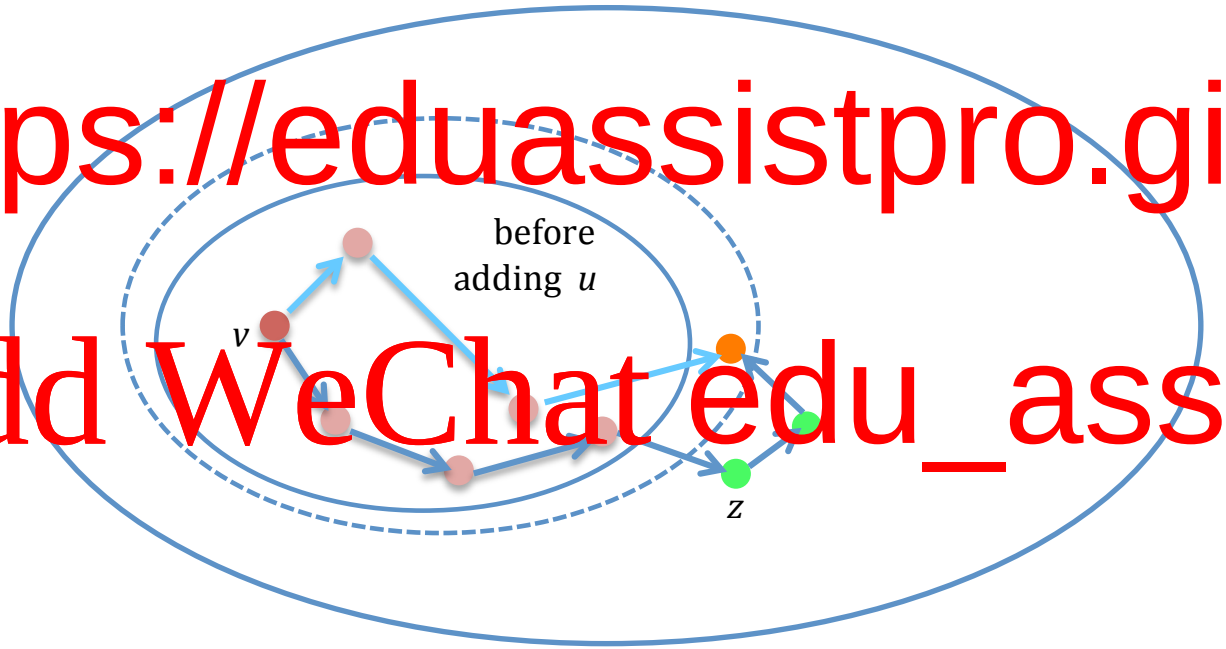
Dijkstra's Shortest Paths Algorithm

- The algorithm builds a set S of vertices for which the shortest path has been already established, starting with a single source vertex $S = \{v\}$ and adding one vertex at a time.
- At each stage of the construction, we add the element $u \in V \setminus S$ which has the shortest path from v to u with all intermediate vertices already in S.



Dijkstra's Shortest Paths Algorithm

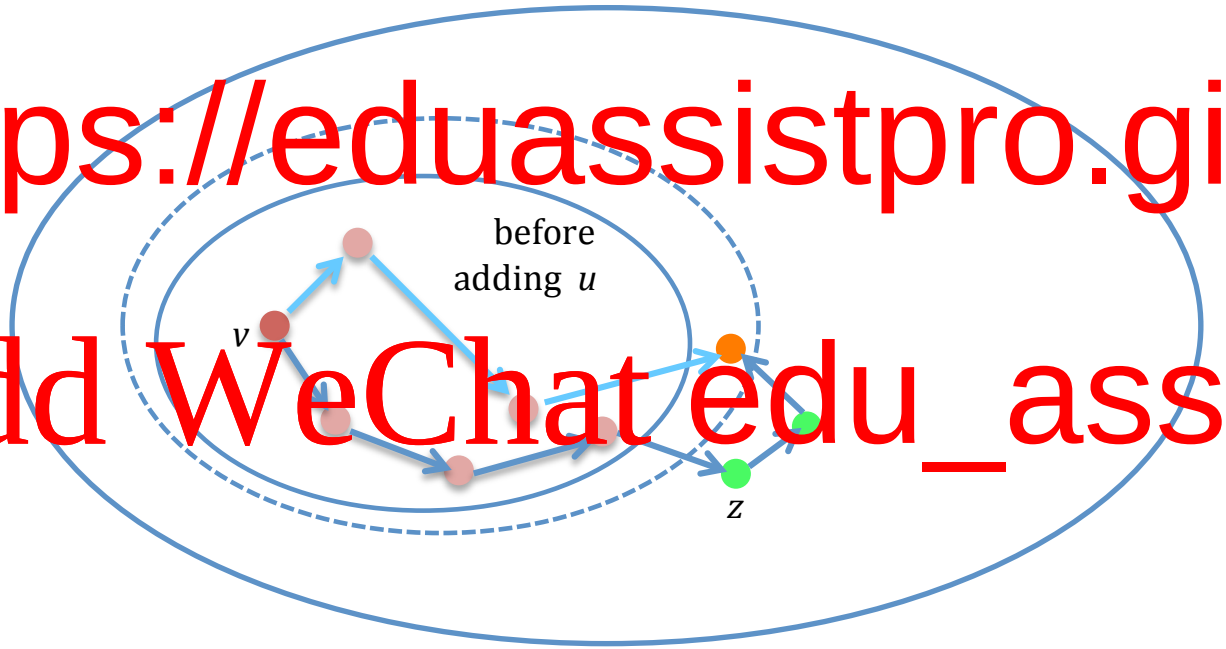
- Why does this produce a shortest path from v to u in G?
- Assume the opposite, that there exists a shorter path from v to u in G.
- By our choice of u such a path cannot be entirely in S.
- Let z be the first vertex outside S (as it was just prior to addition of u) on su



- Since there are no negative weight edges the path from v to such z would be shorter than the path from v to u, contradicting our choice of u.

Dijkstra's Shortest Paths Algorithm

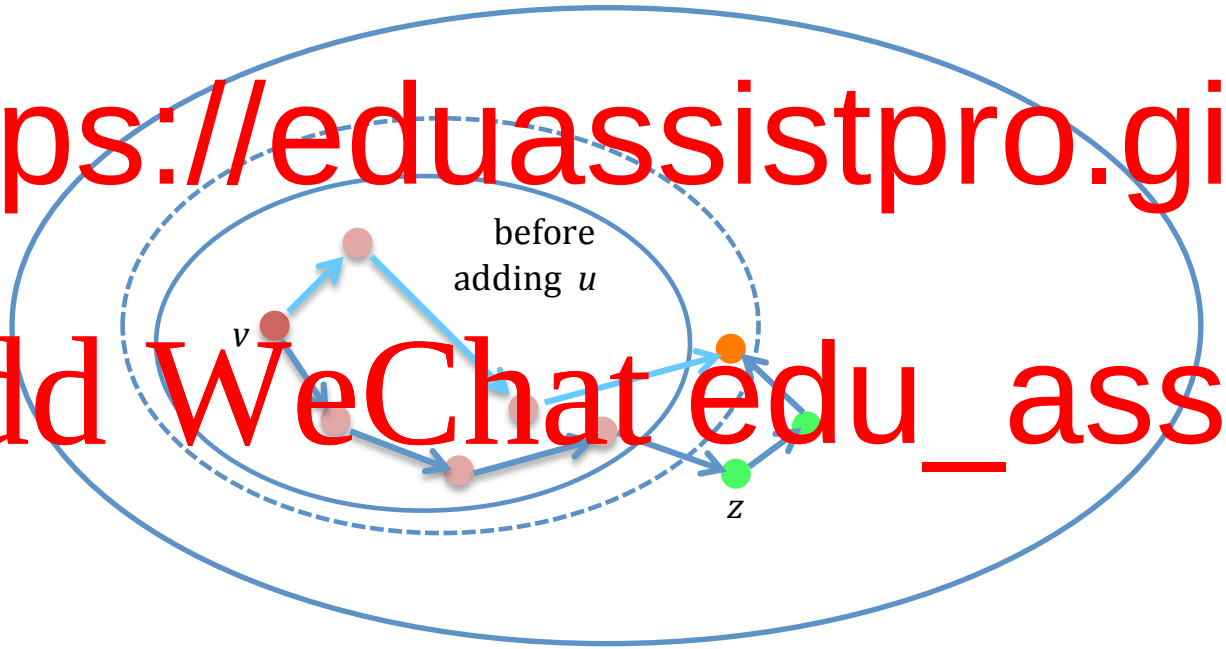
- Why does this produce a shortest path from v to u in G?
- Assume the opposite, that there exists a shorter path from v to u in G. By our choice of u such a path cannot be entirely in S.
- Let z be the first vertex outside S (as it was just prior to addition of u) on su



- Since there are no negative weight edges the path from v to such z would be shorter than the path from v to u, contradicting our choice of u.

Dijkstra's Shortest Paths Algorithm

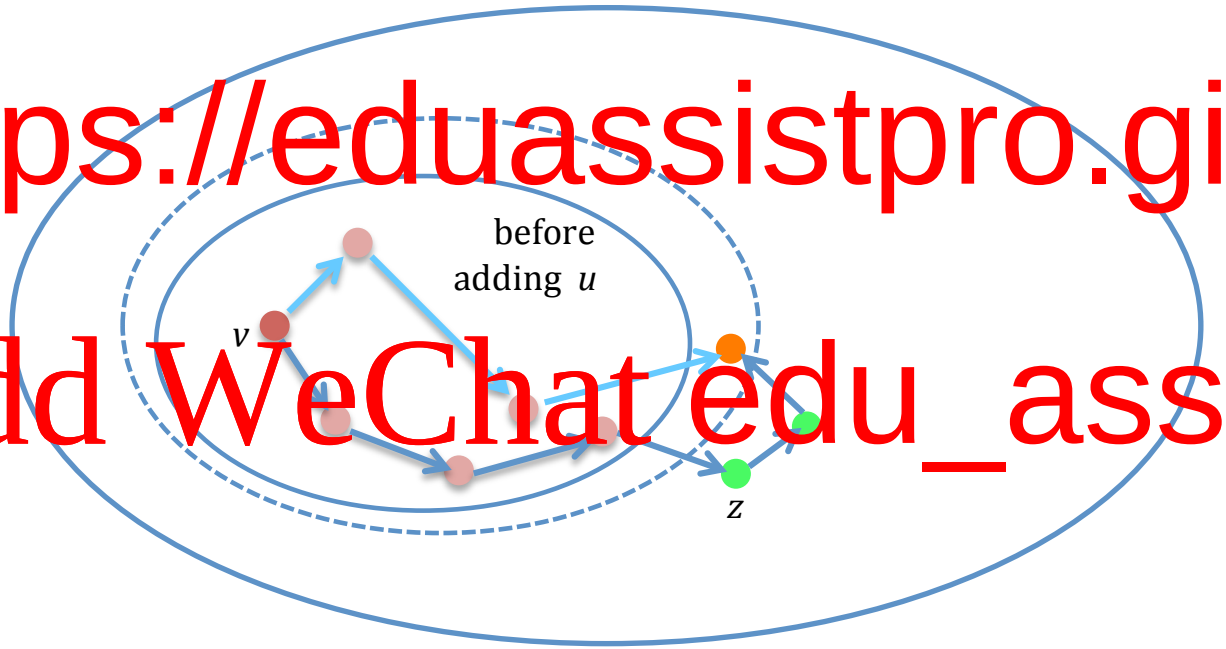
- Why does this produce a shortest path from v to u in G?
- Assume the opposite, that there exists a shorter path from v to u in G. By our choice of u such a path cannot be entirely in S.
- Let z be the first vertex outside S (as it was just prior to addition of u) on su



- Since there are no negative weight edges the path from v to such z would be shorter than the path from v to u, contradicting our choice of u.

Dijkstra's Shortest Paths Algorithm

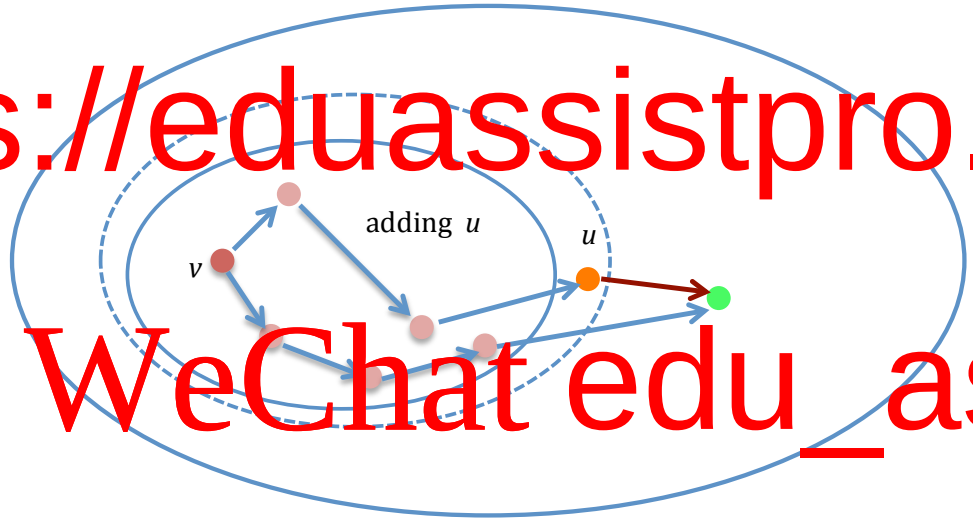
- Why does this produce a shortest path from v to u in G?
- Assume the opposite, that there exists a shorter path from v to u in G. By our choice of u such a path cannot be entirely in S.
- Let z be the first vertex outside S (as it was just prior to addition of u) on su



- Since there are no negative weight edges the path from v to such z would be shorter than the path from v to u, contradicting our choice of u.

Dijkstra's Shortest Paths Algorithm

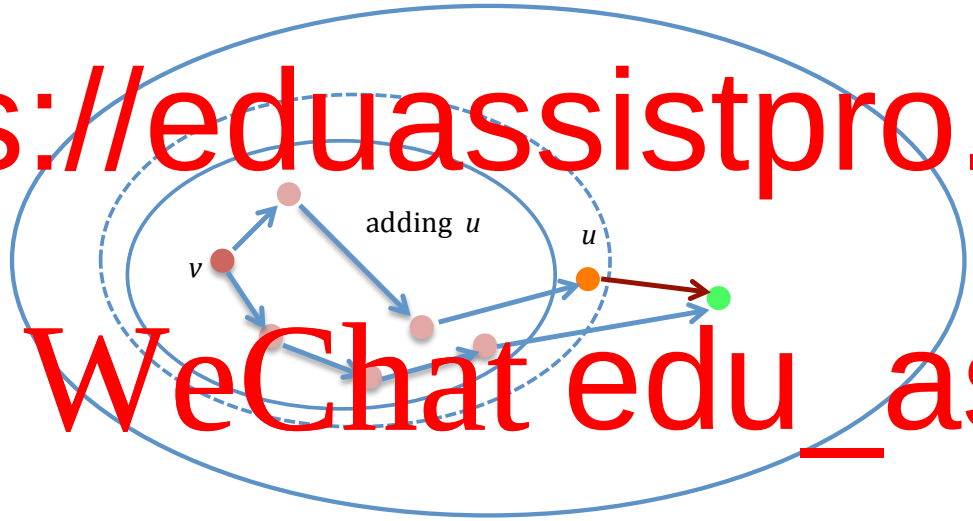
- How is this construction done efficiently?
- Initially all vertices except v are placed in a heap based priority queue with additional Position array, with the weight $w(v, u)$ if $(v, u) \in E$ or ∞ as the key;
- As we progress, the key of each element u will be updated with length $\text{lhs}_{S,v}(u)$ of the shortest path from v to u which has all intermediate vertices on such a path in S.



- We always pop the element u from the priority queue which has the smallest key and add it to set S.
- We then look for all elements $z \in V \setminus S$ for which $(u, z) \in E$ and if $\text{lhs}_{S,v}(u) + w(u, z) < \text{lhs}_{S,v}(z)$ we update the key of z to the value $\text{lhs}_{S,v}(u) + w(u, z)$.

Dijkstra's Shortest Paths Algorithm

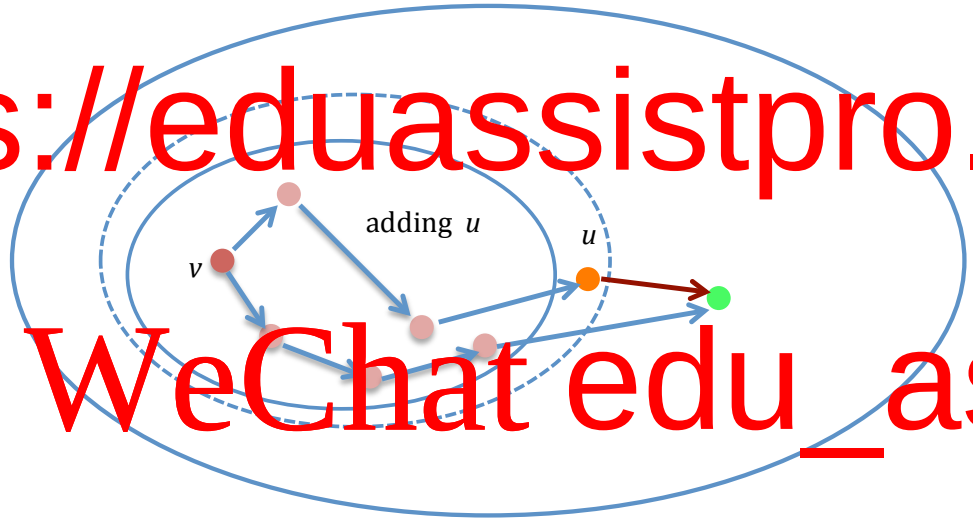
- How is this construction done efficiently?
- Initially all vertices except v are placed in a heap based priority queue with additional Position array, with the weight $w(v, u)$ if $(v, u) \in E$ or ∞ as the key;
- As we progress, the key of each element u will be updated with length $\text{lhs}_{S,v}(u)$ of the shortest path from v to u which has all intermediate vertices on such a path in S.



- We always pop the element u from the priority queue which has the smallest key and add it to set S.
- We then look for all elements $z \in V \setminus S$ for which $(u, z) \in E$ and if $\text{lhs}_{S,v}(u) + w(u, z) < \text{lhs}_{S,v}(z)$ we update the key of z to the value $\text{lhs}_{S,v}(u) + w(u, z)$.

Dijkstra's Shortest Paths Algorithm

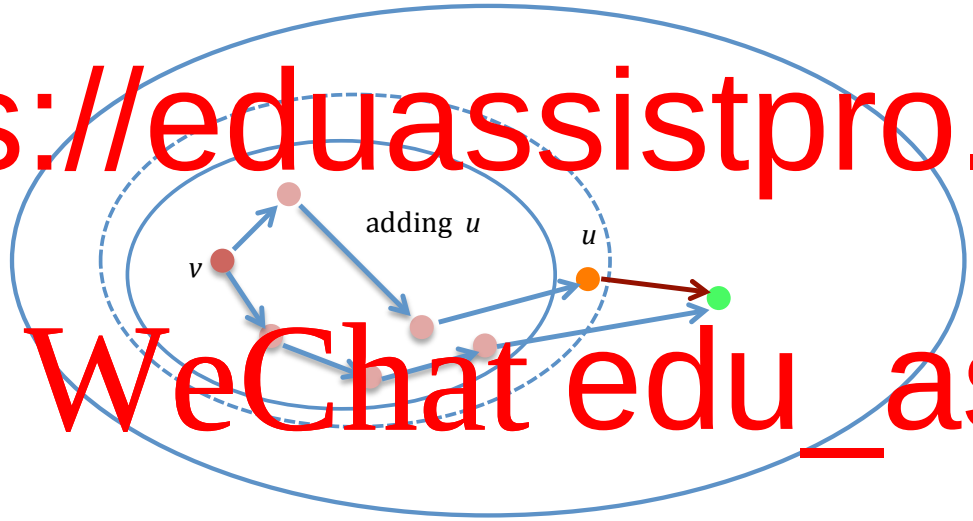
- How is this construction done efficiently?
- Initially all vertices except v are placed in a heap based priority queue with additional Position array, with the weight $w(v, u)$ if $(v, u) \in E$ or ∞ as the key;
- As we progress, the key of each element u will be updated with length $\text{lhs}_{S,v}(u)$ of the shortest path from v to u which has all intermediate vertices on such a path in S.



- We always pop the element u from the priority queue which has the smallest key and add it to set S.
- We then look for all elements $z \in V \setminus S$ for which $(u, z) \in E$ and if $\text{lhs}_{S,v}(u) + w(u, z) < \text{lhs}_{S,v}(z)$ we update the key of z to the value $\text{lhs}_{S,v}(u) + w(u, z)$.

Dijkstra's Shortest Paths Algorithm

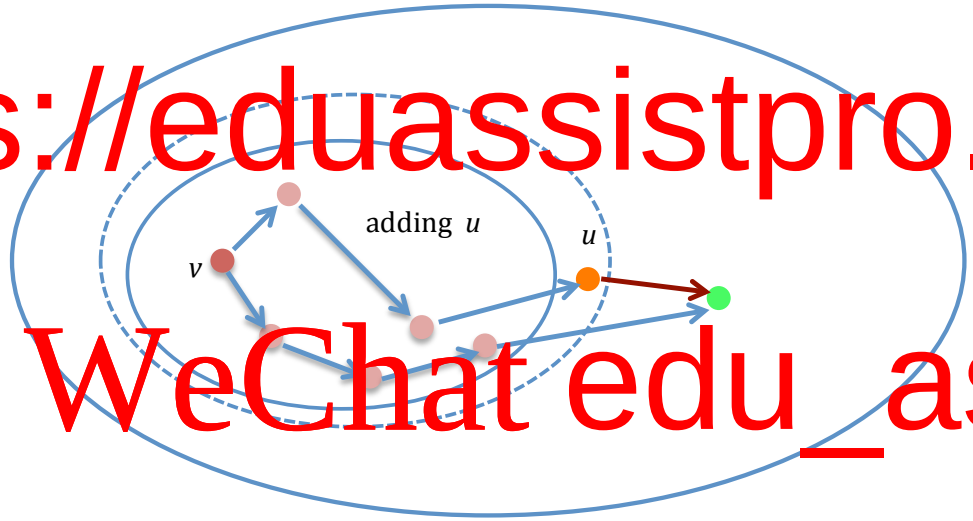
- How is this construction done efficiently?
- Initially all vertices except v are placed in a heap based priority queue with additional Position array, with the weight $w(v,u)$ if $(v,u) \in E$ or ∞ as the key;
- As we progress, the key of each element u will be updated with length $\text{lhs}_{S,v}(u)$ of the shortest path from v to u which has all intermediate vertices on such a path in S.



- We always pop the element u from the priority queue which has the smallest key and add it to set S.
- We then look for all elements $z \in V \setminus S$ for which $(u,z) \in E$ and if $\text{lhs}_{S,v}(u) + w(u,z) < \text{lhs}_{S,v}(z)$ we update the key of z to the value $\text{lhs}_{S,v}(u) + w(u,z)$.

Dijkstra's Shortest Paths Algorithm

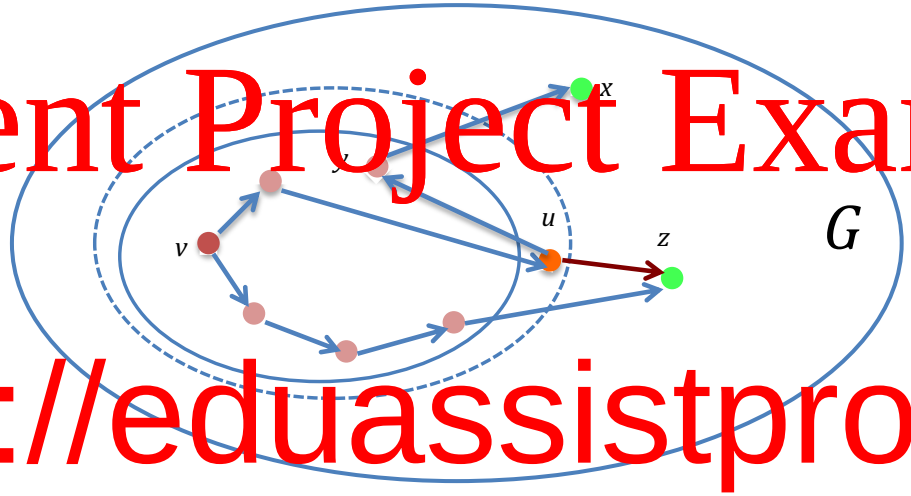
- How is this construction done efficiently?
- Initially all vertices except v are placed in a heap based priority queue with additional Position array, with the weight $w(v, u)$ if $(v, u) \in E$ or ∞ as the key;
- As we progress, the key of each element u will be updated with length $\text{lh}_{S,v}(u)$ of the shortest path from v to u which has all intermediate vertices on such a path in S.



- We always pop the element u from the priority queue which has the smallest key and add it to set S.
- We then look for all elements $z \in V \setminus S$ for which $(u, z) \in E$ and if $\text{lh}_{S,v}(u) + w(u, z) < \text{lh}_{S,v}(z)$ we update the key of z to the value $\text{lh}_{S,v}(u) + w(u, z)$.

Dijkstra's Shortest Paths Algorithm

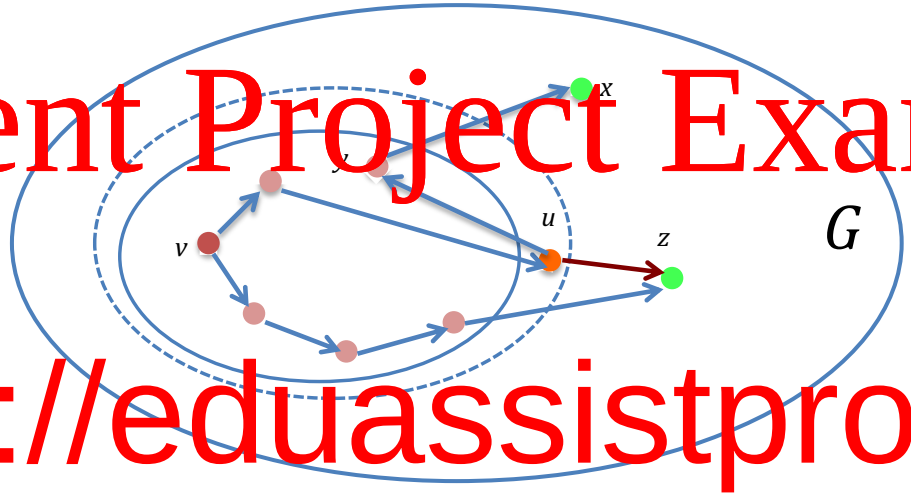
- Why is this enough; i.e., why the only shortest paths which could have changed have u as the last vertex in S?



- Assume opposite that the shortest path to a vertex x has changed when u was added, and that instead of u ano x before x on such a new shortest path.
- However, this is not possible because it would prod y with a vertex u on that path not belonging t g u.
- If graph G has n vertices and m edges, then each edge is inspected only once, when it is an outgoing edge from the most recently added vertex; updating a vertex key takes $O(\log n)$ many steps.
- Thus, in total, the algorithm runs in time $O(m \log n)$.

Dijkstra's Shortest Paths Algorithm

- Why is this enough; i.e., why the only shortest paths which could have changed have u as the last vertex in S?

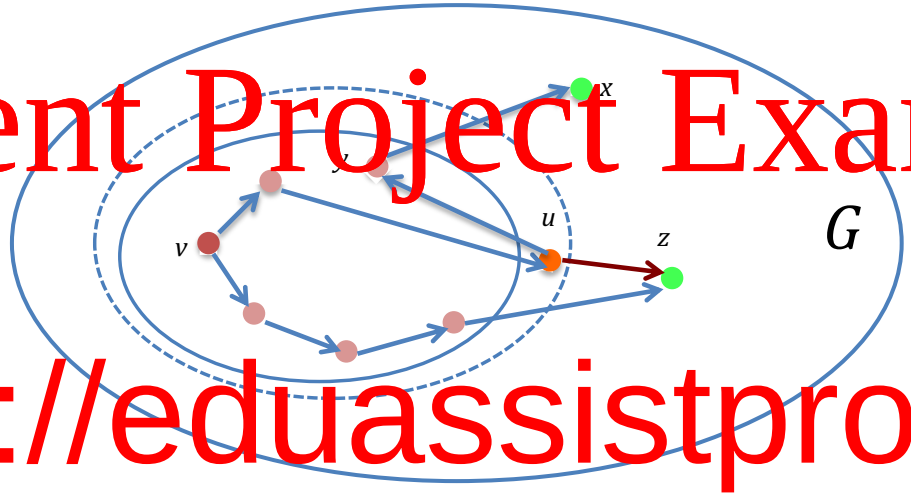


- Assume opposite that the shortest path to a vertex x has changed when u was added, and that instead of u ano x before x on such a new shortest path.
- However, this is not possible because it would prod y with a vertex u on that path not belonging t g u.

- If graph G has n vertices and m edges, then each edge is inspected only once, when it is an outgoing edge from the most recently added vertex; updating a vertex key takes $O(\log n)$ many steps.
- Thus, in total, the algorithm runs in time $O(m \log n)$.

Dijkstra's Shortest Paths Algorithm

- Why is this enough; i.e., why the only shortest paths which could have changed have u as the last vertex in S?



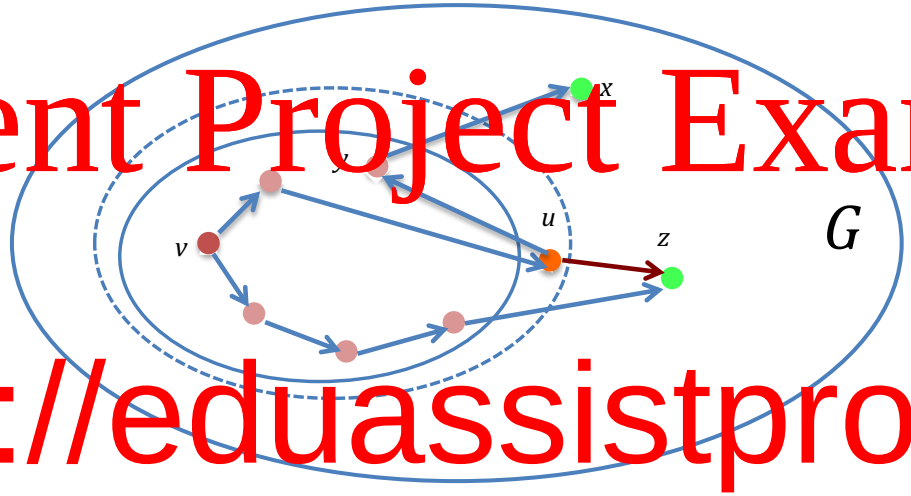
- Assume opposite that the shortest path to a vertex x has changed when u was added, and that instead of u ano x before x on such a new shortest path.
- However, this is not possible because it would prod y with a vertex u on that path not belonging t g u.

- If graph G has n vertices and m edges, then each edge is inspected only once, when it is an outgoing edge from the most recently added vertex; updating a vertex key takes $O(\log n)$ many steps.

- Thus, in total, the algorithm runs in time $O(m \log n)$.

Dijkstra's Shortest Paths Algorithm

- Why is this enough; i.e., why the only shortest paths which could have changed have u as the last vertex in S?



- Assume opposite that the shortest path to a vertex x has changed when u was added, and that instead of u ano x before x on such a new shortest path.
- Howe y with a vertex u on that path not belonging t g u.

- If graph G has n vertices and m edges, then each edge is inspected only once, when it is an outgoing edge from the most recently added vertex; updating a vertex key takes $O(\log n)$ many steps.

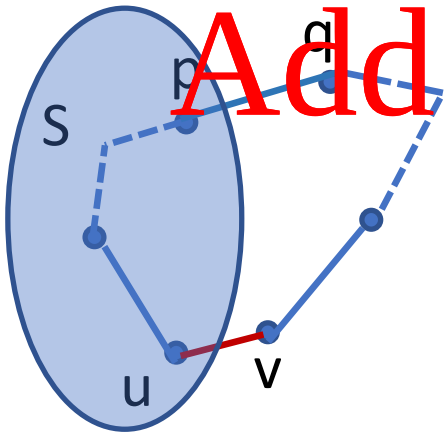
- Thus, in total, the algorithm runs in time $O(m \log n)$.

Greedy Method for graphs: Minimum Spanning Trees

- **Definition:** A minimum spanning tree T of a connected graph G is a subgraph of G (with the same set of vertices) which is a tree, and among all such trees it is of minimal total length of all edges in T.

- **Lemma:** Let G be a connected graph with all lengths of edges E of G dist V of G. As $\notin S$ and is of min a must belo

- **Proof:**



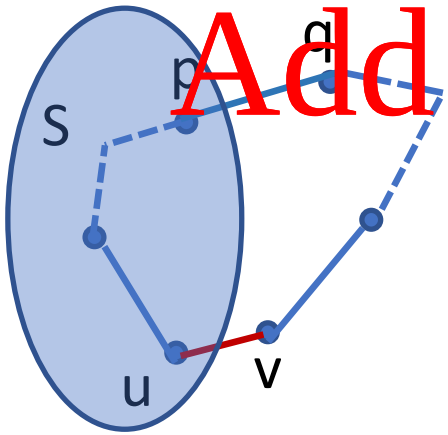
- Assume contrariwise that a minimum spanni contain such an edge $e = (u, v)$.
- Since T is a spanning tree, there exists a path from u to v within such a tree.

Greedy Method for graphs: Minimum Spanning Trees

- **Definition:** A minimum spanning tree T of a connected graph G is a subgraph of G (with the same set of vertices) which is a tree, and among all such trees it is of minimal total length of all edges in T.

- **Lemma:** Let G be a connected graph with all lengths of edges E of G dist V of G. As $\notin S$ and is of min e must belo

- **Proof:**



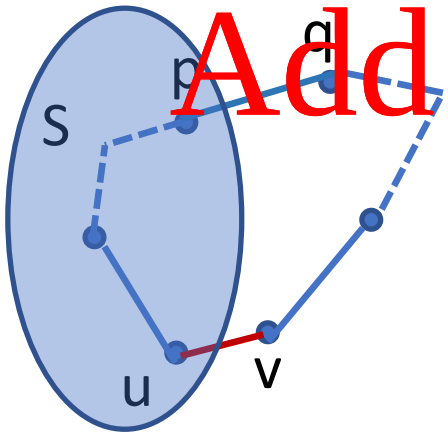
- Assume on the contrary that T is a minimum spanni contain such an edge $e = (u, v)$.
- Since T is a spanning tree, there exists a path from u to v within such a tree.

Greedy Method for graphs: Minimum Spanning Trees

- **Definition:** A minimum spanning tree T of a connected graph G is a subgraph of G (with the same set of vertices) which is a tree, and among all such trees it is of minimal total length of all edges in T.

- **Lemma:** Let G be a connected graph with all lengths of edges E of G dist V of G. As $\notin S$ and is of min e must belo

- **Proof:**



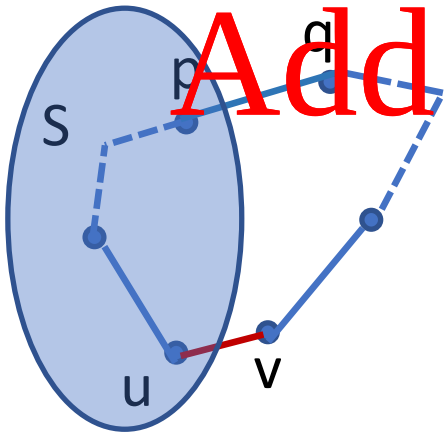
- Assume opposite, t minimum spanni contain such an edge $e = (u, v)$.
- Since T is a spanning tree, there exists a path from u to v within such a tree.

Greedy Method for graphs: Minimum Spanning Trees

- **Definition:** A minimum spanning tree T of a connected graph G is a subgraph of G (with the same set of vertices) which is a tree, and among all such trees it is of minimal total length of all edges in T.

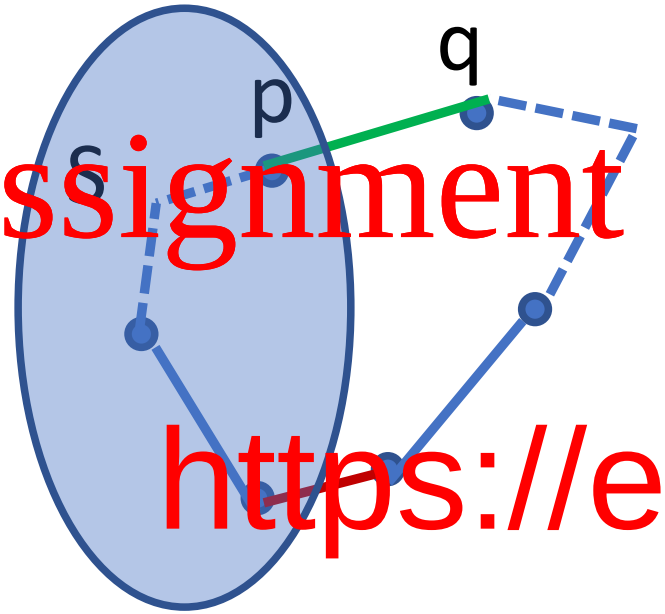
- **Lemma:** Let G be a connected graph with all lengths of edges E of G dist V of G. As $\notin S$ and is of min e must belo

- **Proof:**



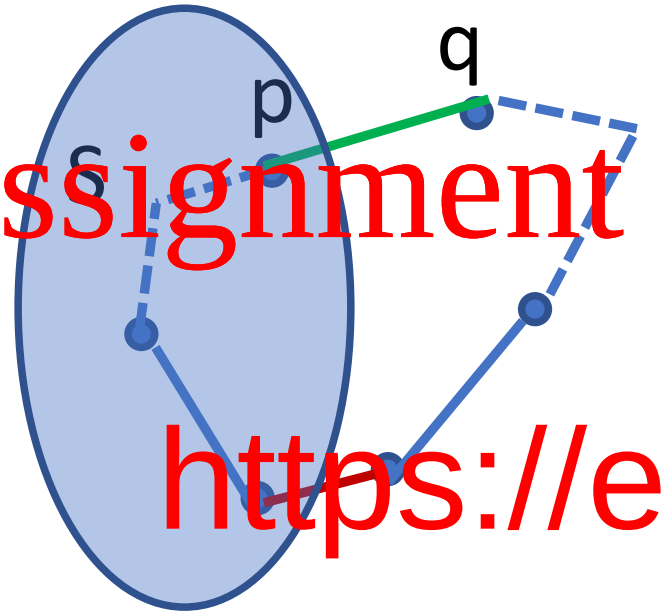
  - Assume opposite, t
    minimum spanni
    contain such an edge $e = (u, v)$.
  - Since T is a spanning tree, there exists a path from u to v within such a tree.

Greedy Method for graphs: Minimum Spanning Trees



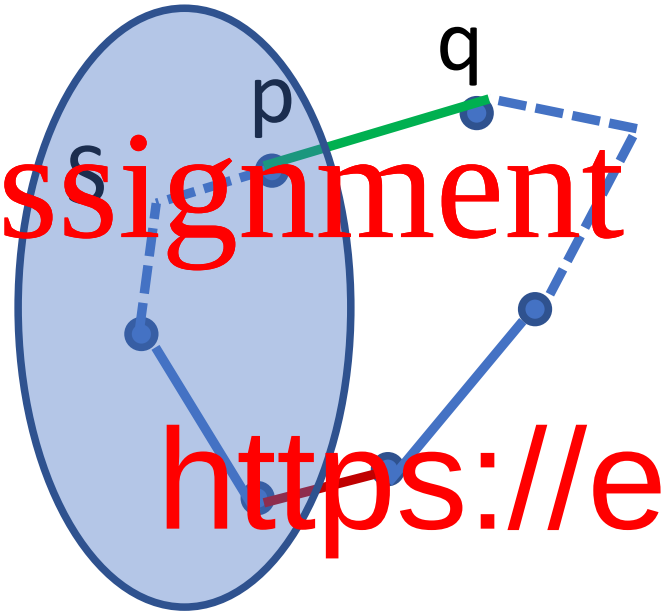
- Since $u \in S$ and $v \notin S$, this path must have left S at a certain point.
- Assume that p is the last vertex on this path which is in S and $q \notin S$ the vertex immediately after p on that path.
- However, the edge (p, q) belongs to T and must have a length larger than the edge (u, v) (which is the minimal length edge with one end in S and one end outside S.
- o
  ohoices
- Howe
  weigh
  than th
  would r
  smaller weight, contradicting our assumption that T is a minimum spanning tree.

Greedy Method for graphs: Minimum Spanning Trees



- Since $u \in S$ and $v \notin S$, this path must have left S at a certain point.
- Assume that p is the last vertex on this path which is in S and $q \notin S$ the vertex immediately after p on that path.

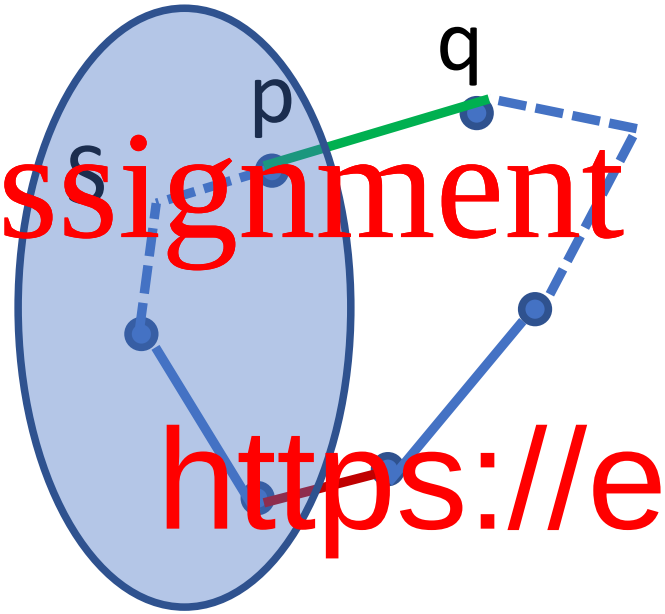
Greedy Method for graphs: Minimum Spanning Trees



- However, the edge (p, q) belongs to T and must have a length larger than the edge (u, v) (which is the minimal length edge with one end in S and one end outside S.
- o
  oduces
- Howe
  weigh
  than th
  would r
  smaller weight, contradicting our assumption that T is a minimum spanning tree.

- Since $u \in S$ and $v \notin S$, this path must have left S at a certain point.
- Assume that p is the last vertex on this path which is in S and $q \notin S$ the vertex immediately after p on that path.

Greedy Method for graphs: Minimum Spanning Trees

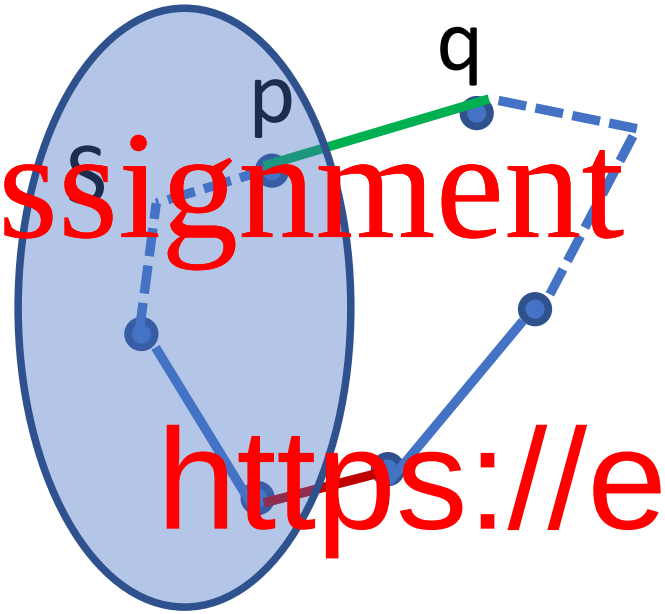


- However, the edge (p, q) belongs to T and must have a length larger than the edge (u, v) (which is the minimal length edge with one end in S and one end outside S.
- o
oduces

- Since $u \in S$ and $v \notin S$, this path must have left S at a certain point.
- Assume that p is the last vertex on this path which is in S and $q \notin S$ the vertex immediately after p on that path.

- Howe
weigh
than th
would r
smaller weight, contradicting our assumption that T is a minimum spanning tree.

Greedy Method for graphs: Minimum Spanning Trees



- Since $u \in S$ and $v \notin S$, this path must have left S at a certain point.
- Assume that p is the last vertex on this path which is in S and $q \notin S$ the vertex immediately after p on that path.

- However, the edge (p, q) belongs to T and must have a length larger than the edge (u, v) (which is the minimal length edge with one end in S and one end outside S.
- o
  oduces
- Howe
  weigh
  than th s
  would r
  smaller weight, contradicting our assumption that T is a minimum spanning tree.

The Kruskal Algorithm

- We order the edges E in a non-decreasing order with respect to their wei
- We
- An edge e_i is added at a round i of construction whenever its addition does not introduce a cycle in the graph constructed t
- If adding an edge does introduce a cycle, that edge is di
- The process terminates when the list of all edges has been exhausted.

The Kruskal Algorithm

- We order the edges E in a non-decreasing order with respect to their wei
- We
- An edge e_i is added at a round i of construction whenever its addition does not introduce a cycle in the graph constructed t
- If adding an edge does introduce a cycle, that edge is di
- The process terminates when the list of all edges has been exhausted.

Greedy Method for graphs: Minimum Spanning Trees

The Kruskal Algorithm

- We order the edges E in a non-decreasing order with respect to their wei
- We b

- An edge e_i is added at a round i of construction whenever its addition does not introduce a cycle in the graph constructed t
- If adding an edge does introduce a cycle, that edge is di
- The process terminates when the list of all edges has been exhausted.

Greedy Method for graphs: Minimum Spanning Trees

The Kruskal Algorithm

- We order the edges E in a non-decreasing order with respect to their wei
- We b
- An edge e_i is added at a round i of construction whenever its addition does not introduce a cycle in the graph constructed t
- If adding an edge does introduce a cycle, that edge is di
- The process terminates when the list of all edges has been exhausted.

The Kruskal Algorithm

- We order the edges E in a non-decreasing order with respect to their wei
- We b
- An edge e_i is added at a round i of construction whenever its addition does not introduce a cycle in the graph constructed t
- If adding an edge does introduce a cycle, that edge is di
- The process terminates when the list of all edges has been exhausted.

The Kruskal Algorithm

- We order the edges E in a non-decreasing order with respect to their wei
- We b
- An edge e_i is added at a round i of construction whenever its addition does not introduce a cycle in the graph constructed t
- If adding an edge does introduce a cycle, that edge is di
- The process terminates when the list of all edges has been exhausted.

Minimum Spanning Trees: the Kruskal Algorithm

Claim: The Kruskal algorithm produces a minimal spanning tree, and if all weights are distinct, then such a Minimum Spanning Tree is unique.

Proof: We consider the case when all weights are distinct.

- Consider an arbitrary edge $e = (u, v)$ added in the course of the Kruskal algorithm.
- Con
to
algo
- The
- Until this moment no edge with one end in
has been considered because otherwise it would h
formi a cycle.
- Consequently, edge $e = (u, v)$ is the shortest
and by the previous lemma it must belong to every sp
- Thus, the set of edges produced by the Kruskal algorithm is a subset of the set of edges of every spanning tree.
- But the structure produced by the Kruskal algorithm is clearly cycle-free and it spans the graph, otherwise another edge could be added.

Minimum Spanning Trees: the Kruskal Algorithm

Claim: The Kruskal algorithm produces a minimal spanning tree, and if all weights are distinct, then such a Minimum Spanning Tree is unique.

Proof: We consider the case when all weights are distinct.

- Consider an arbitrary edge $e = (u, v)$ added in the course of the Kruskal algorithm.
- Con
to
algo
- The
- Until this moment no edge with one end in
has been considered because otherwise it would h
formi
- Consequently, edge $e = (u, v)$ is the shortest
and by the previous lemma it must belong to every sp
- Thus, the set of edges produced by the Kruskal algorithm is a subset of the set of edges of every spanning tree.
- But the structure produced by the Kruskal algorithm is clearly cycle-free and it spans the graph, otherwise another edge could be added.

Minimum Spanning Trees: the Kruskal Algorithm

Claim: The Kruskal algorithm produces a minimal spanning tree, and if all weights are distinct, then such a Minimum Spanning Tree is unique.

Proof: We consider the case when all weights are distinct.

- Consider an arbitrary edge $e = (u, v)$ added in the course of the Kruskal algorithm.
- Con
to
algo
- The
- Until this moment no edge with one end in
has been considered because otherwise it would h
formi a cycle.
- Consequently, edge $e = (u, v)$ is the shortest
and by the previous lemma it must belong to every sp
- Thus, the set of edges produced by the Kruskal algorithm is a subset of the set of edges of every spanning tree.
- But the structure produced by the Kruskal algorithm is clearly cycle-free and it spans the graph, otherwise another edge could be added.

Minimum Spanning Trees: the Kruskal Algorithm

Claim: The Kruskal algorithm produces a minimal spanning tree, and if all weights are distinct, then such a Minimum Spanning Tree is unique.

Proof: We consider the case when all weights are distinct.

- Consider an arbitrary edge $e = (u, v)$ added in the course of the Kruskal algorithm.
- Con
to
algo
- The
- Until this moment no edge with one end in S has been considered because otherwise it would h
formi a cycle.
- Consequently, edge $e = (u, v)$ is the shortest
and by the previous lemma it must belong to every sp
- Thus, the set of edges produced by the Kruskal algorithm is a subset of the set of edges of every spanning tree.
- But the structure produced by the Kruskal algorithm is clearly cycle-free and it spans the graph, otherwise another edge could be added.

Minimum Spanning Trees: the Kruskal Algorithm

Claim: The Kruskal algorithm produces a minimal spanning tree, and if all weights are distinct, then such a Minimum Spanning Tree is unique.

Proof: We consider the case when all weights are distinct.

- Consider an arbitrary edge $e = (u, v)$ added in the course of the Kruskal algorithm.
- Con
to
algo
- The
- Until this moment no edge with one end in S has been considered because otherwise it would h
formin a cycle.
- Consequently, edge $e = (u, v)$ is the shortest
and by the previous lemma it must belong to every sp
- Thus, the set of edges produced by the Kruskal algorithm is a subset of the set of edges of every spanning tree.
- But the structure produced by the Kruskal algorithm is clearly cycle-free and it spans the graph, otherwise another edge could be added.

Minimum Spanning Trees: the Kruskal Algorithm

Claim: The Kruskal algorithm produces a minimal spanning tree, and if all weights are distinct, then such a Minimum Spanning Tree is unique.

Proof: We consider the case when all weights are distinct.

- Consider an arbitrary edge $e = (u, v)$ added in the course of the Kruskal algorithm.
- Con
to
algo
- The
- Until this moment no edge with one end in S has been considered because otherwise it would h
forming a cycle.
- Consequently, edge $e = (u, v)$ is the shortest
and by the previous lemma it must belong to every sp
- Thus, the set of edges produced by the Kruskal algorithm is a subset of the set of edges of every spanning tree.
- But the structure produced by the Kruskal algorithm is clearly cycle-free and it spans the graph, otherwise another edge could be added.

Minimum Spanning Trees: the Kruskal Algorithm

Claim: The Kruskal algorithm produces a minimal spanning tree, and if all weights are distinct, then such a Minimum Spanning Tree is unique.

Proof: We consider the case when all weights are distinct.

- Consider an arbitrary edge $e = (u, v)$ added in the course of the Kruskal algorithm.
- Con
to
algo
- The
- Until this moment no edge with one end in
has been considered because otherwise it would h
forming a cycle.
- Consequently, edge $e = (u, v)$ is the shortest
and by the previous lemma it must belong to every sp
- Thus, the set of edges produced by the Kruskal algorithm is a subset of the set of edges of every spanning tree.
- But the structure produced by the Kruskal algorithm is clearly cycle-free and it spans the graph, otherwise another edge could be added.

Minimum Spanning Trees: the Kruskal Algorithm

Claim: The Kruskal algorithm produces a minimal spanning tree, and if all weights are distinct, then such a Minimum Spanning Tree is unique.

Proof: We consider the case when all weights are distinct.

- Consider an arbitrary edge $e = (u, v)$ added in the course of the Kruskal algorithm.
- Con
to
algo
- The
- Until this moment no edge with one end in
has been considered because otherwise it would h
forming a cycle.
- Consequently, edge $e = (u, v)$ is the shortest
and by the previous lemma it must belong to every sp
- Thus, the set of edges produced by the Kruskal algorithm is a subset of the set of edges of every spanning tree.
- But the structure produced by the Kruskal algorithm is clearly cycle-free and it spans the graph, otherwise another edge could be added.

Minimum Spanning Trees: the Kruskal Algorithm

Claim: The Kruskal algorithm produces a minimal spanning tree, and if all weights are distinct, then such a Minimum Spanning Tree is unique.

Proof: We consider the case when all weights are distinct.

- Consider an arbitrary edge $e = (u, v)$ added in the course of the Kruskal algorithm.
- Con
to
algo
- The
- Until this moment no edge with one end in
has been considered because otherwise it would h
forming a cycle.
- Consequently, edge $e = (u, v)$ is the shortest
and by the previous lemma it must belong to every sp
- Thus, the set of edges produced by the Kruskal algorithm is a subset of the set of edges of every spanning tree.
- But the structure produced by the Kruskal algorithm is clearly cycle-free and it spans the graph, otherwise another edge could be added.

Efficient implementation of the Kruskal Algorithm

- To efficiently implement the Kruskal algorithm, we need a useful data structure for storing sets of elements, called the Union-Find.

Such a data structure has to support three operations:

1. MakeUnionFind(S), which, given a set S returns a strucure in
2. et to which v belongs. Such operation should (1) or time $O(\log n)$ as we explain later.
3. Union(A, B), which, given (the labels of) t nges the data structure by replacing sets A and B with the set $A \cup B$. A sequence of k consecutive Union operations should run in time $O(k \log k)$.

Efficient implementation of the Kruskal Algorithm

- To efficiently implement the Kruskal algorithm, we need a useful data structure for storing sets of elements, called the Union-Find.
- Such a data structure has to support three operations

1. MakeUnionFind(S), which, given a set S returns a strucure in
2. which v belongs. Such operation should (1) or time $O(\log n)$ as we explain later.
3. Union(A, B), which, given (the labels of) t nges the data structure by replacing sets A and B with the set $A \cup B$. A sequence of k consecutive Union operations should run in time $O(k \log k)$.

Efficient implementation of the Kruskal Algorithm

- To efficiently implement the Kruskal algorithm, we need a useful data structure for storing sets of elements, called the Union-Find.
- Such a data structure has to support three operations
  1. MakeUnionFind(S), which, given a set S returns a strucure in
  2. Find(v), which, given a vertex v, returns the label of the set to which v belongs. Such operation should run in time $O(1)$ or time $O(\log n)$ as we explain later.
  3. Union(A, B), which, given (the labels of) two sets A and B, changes the data structure by replacing sets A and B with the set $A \cup B$. A sequence of k consecutive Union operations should run in time $O(k \log k)$.

Efficient implementation of the Kruskal Algorithm

- To efficiently implement the Kruskal algorithm, we need a useful data structure for storing sets of elements, called the Union-Find.
- Such a data structure has to support three operations:
  1. MakeUnionFind(S), which, given a set S returns a strucure in
  2. which v belongs. Such operation should et to (1) or time $O(\log n)$ as we explain later.
  3. Union(A, B), which, given (the labels of) t nges the data structure by replacing sets A and B with the set $A \cup B$. A sequence of k consecutive Union operations should run in time $O(k \log k)$.

Efficient implementation of the Kruskal Algorithm

- To efficiently implement the Kruskal algorithm, we need a useful data structure for storing sets of elements, called the Union-Find.
- Such a data structure has to support three operations:
  1. MakeUnionFind(S), which, given a set S returns a strucure in
  2. et to which v belongs. Such operation should (1) or time $O(\log n)$ as we explain later.
  3. Union(A, B), which, given (the labels of) t nges the data structure by replacing sets A and B with the set $A \cup B$. A sequence of k consecutive Union operations should run in time $O(k \log k)$.

Efficient implementation of the Kruskal Algorithm

- Note that we do not give the run time of a single UNION operation but of a sequence of k consecutive such operations.
- Such time complexity analysis is called *amortized analysis*; it (essentially) estimates average cost of an operation in a sequence of operations, in this case $\log k$.
- We Uni can a $2, \ldots, n\}$.
- The simplest implementation of the Union-Fin of:
  1. an array A such that $A[i] = j$ mea j;
  2. an array B such that $B[i]$ contains the number of elements in the set labeled by i (which can be 0) and pointers to the first and last element of a linked list of elements of the set labeled by i.

Efficient implementation of the Kruskal Algorithm

- Note that we do not give the run time of a single UNION operation but of a sequence of k consecutive such operations.
- Such time complexity analysis is called *amortized analysis*; it (essentially) estimates average cost of an operation in a sequence of operations, in this case $\log k$.
- We Uni can a $2, \ldots, n\}$.
- The simplest implementation of the Union-Fin of:
  1. an array A such that $A[i] = j$ mea j;
  2. an array B such that $B[i]$ contains the number of elements in the set labeled by i (which can be 0) and pointers to the first and last element of a linked list of elements of the set labeled by i.

Efficient implementation of the Kruskal Algorithm

- Note that we do not give the run time of a single UNION operation but of a sequence of k consecutive such operations.
- Such time complexity analysis is called *amortized analysis*; it (essentially) estimates average cost of an operation in a sequence of operations, in this case $\log k$.
- We
  Uni
  can a $2, \ldots, n\}$.
- The simplest implementation of the Union-Fin
  of:
  1. an array A such that $A[i] = j$ mea j;
  2. an array B such that $B[i]$ contains the number of elements in the set labeled by i (which can be 0) and pointers to the first and last element of a linked list of elements of the set labeled by i.

Efficient implementation of the Kruskal Algorithm

- Note that we do not give the run time of a single UNION operation but of a sequence of k consecutive such operations.
- Such time complexity analysis is called *amortized analysis*; it (essentially) estimates average cost of an operation in a sequence of operations, in this case $\log k$.
- We
  Uni
  can a $2, \ldots, n\}$.
- The simplest implementation of the Union-Fin
  of:
  1. an array A such that $A[i] = j$ mea j;
  2. an array B such that $B[i]$ contains the number of elements in the set labeled by i (which can be 0) and pointers to the first and last element of a linked list of elements of the set labeled by i.

Efficient implementation of the Kruskal Algorithm

- Note that we do not give the run time of a single UNION operation but of a sequence of k consecutive such operations.
- Such time complexity analysis is called *amortized analysis*; it (essentially) estimates average cost of an operation in a sequence of operations, in this case $\log k$.
- We
  Uni
  can a $2, \ldots, n\}$.
- The simplest implementation of the Union-Fin
  of:
  1. an array A such that $A[i] = j$ mea j;
  2. an array B such that $B[i]$ contains the number of elements in the set labeled by i (which can be 0) and pointers to the first and last element of a linked list of elements of the set labeled by i.

Efficient implementation of the Kruskal Algorithm

- UNION(i,j) of two sets labeled by i and j, respectively, is defined as follows:
  - if number of elements in the set labeled by i is larger or equal to the number of elements in the set labeled by j then labels in array A of rray B is
  - i, then the labels of all elements in the set labeled b ged to j and array B is updated accordingly
- Note that this definition implies that if f the set containing an element m, then the new set containing m will have at least twice the number of elements of the set which contained m before the UNION(i, j) operation.

Efficient implementation of the Kruskal Algorithm

- UNION(i,j) of two sets labeled by i and j, respectively, is defined as follows:
  - if number of elements in the set labeled by i is larger or equal to the number of elements in the set labeled by j then labels in array A of rray B is
  - i, then the labels of all elements in the set labeled b ged to j and array B is updated accordingly
- Note that this definition implies that if f the set containing an element m, then the new set containing m will have at least twice the number of elements of the set which contained m before the UNION(i, j) operation.

Efficient implementation of the Kruskal Algorithm

- UNION(i,j) of two sets labeled by i and j, respectively, is defined as follows:
  - if number of elements in the set labeled by i is larger or equal to the number of elements in the set labeled by j then labels in array A of rray B is
  - i, then the labels of all elements in the set labeled b ged to j and array B is updated accordingly
- Note that this definition implies that if f the set containing an element m, then the new set containing m will have at least twice the number of elements of the set which contained m before the UNION(i, j) operation.

Efficient implementation of the Kruskal Algorithm

- Any sequence of k initial consecutive UNION operations can touch at most $2k$ elements of S (which happens if all UNION operations were applied to singleton sets).
- Thu ive UNION mus
- Sinc containing m at least doubles the size of the set containing that element, the label of the set containing m could c many times.
- Thus, since we have at most $2k$ elemen l consecutive UNION operations will have in total fewer than $2k \log 2k$ many label changes in A and each UNION operation changes just a few pointers in B and adds up the sizes of sets.

Efficient implementation of the Kruskal Algorithm

- Any sequence of k initial consecutive UNION operations can touch at most $2k$ elements of S (which happens if all UNION operations were applied to singleton sets).
- Thu ive UNION mus
- Sinc
containing m at least doubles the size of the set containing that element, the label of the set containing m could c many times.
- Thus, since we have at most $2k$ elemen l consecutive UNION operations will have in total fewer than $2k \log 2k$ many label changes in A and each UNION operation changes just a few pointers in B and adds up the sizes of sets.

Efficient implementation of the Kruskal Algorithm

- Any sequence of k initial consecutive UNION operations can touch at most $2k$ elements of S (which happens if all UNION operations were applied to singleton sets).
- Thu ive UNION mus
- Sinc containing m at least doubles the size of the set containing that element, the label of the set containing m could c many times.
- Thus, since we have at most $2k$ elemen l consecutive UNION operations will have in total fewer than $2k \log 2k$ many label changes in A and each UNION operation changes just a few pointers in B and adds up the sizes of sets.

Efficient implementation of the Kruskal Algorithm

- Any sequence of k initial consecutive UNION operations can touch at most $2k$ elements of S (which happens if all UNION operations were applied to singleton sets).
- Thu ive UNION mus
- Sinc containing m at least doubles the size of the set containing that element, the label of the set containing m could c many times.
- Thus, since we have at most $2k$ elemen l consecutive UNION operations will have in total fewer than $2k \log 2k$ many label changes in A and each UNION operation changes just a few pointers in B and adds up the sizes of sets.

Efficient implementation of the Kruskal Algorithm

- Thus, every sequence of k initial consecutive UNION operations has time complexity of $O(k \log k)$.
- Suc
poss
- See the textbook for an Union-Find data structur
path compression, which further reduces the am
UNIO operation to the cost of increasing the co
operation from $O(\ )$ to $O(\log)$.

Efficient implementation of the Kruskal Algorithm

- Thus, every sequence of k initial consecutive UNION operations has time complexity of $O(k \log k)$.
- Suc
poss
- See the textbook for an Union-Find data structur
path compression, which further reduces the am
UNIO operation to the cost of increasing the co AND
operation from $O(\quad)$ to $O(\log n)$.

Efficient implementation of the Kruskal Algorithm

- Thus, every sequence of k initial consecutive UNION operations has time complexity of $O(k \log k)$.

- Suc
poss

- See the textbook for an Union-Find data structur
path compression, which further reduces the am
UNION operation at the cost of increasing the co
operation from $O(1)$ to $O(\log n)$.

Efficient implementation of the Kruskal Algorithm

- We now use the previously described Union-Find data structure to efficiently implement the Kruskal algorithm on a graph $G = (V, E)$ with n vertices and m edges.
- We first have to sort m edges of graph G which takes time $O(m \log m)$. Sinc
  $O($
- As w
  making connected components which will be me
  connected components until all vertices belong t
  component. For this purpose we use Union-Find
  track the connected components constructed t
- For each edge $e = (v, u)$ on the sorted list of edges we use two FIND operations, FIND(u) and FIND(v) to determine if vertices u and v belong to the same component.

Efficient implementation of the Kruskal Algorithm

- We now use the previously described Union-Find data structure to efficiently implement the Kruskal algorithm on a graph $G = (V, E)$ with n vertices and m edges.
- We first have to sort m edges of graph G which takes time $O(m \log m)$. Sinc
  $O($
- As w
  making connected components which will be me
  connected components until all vertices belong t
  component. For this purpose we use Union-Find
  track the connected components constructed t
- For each edge $e = (v, u)$ on the sorted list of edges we use two FIND operations, FIND(u) and FIND(v) to determine if vertices u and v belong to the same component.

Efficient implementation of the Kruskal Algorithm

- We now use the previously described Union-Find data structure to efficiently implement the Kruskal algorithm on a graph $G = (V, E)$ with n vertices and m edges.
- We first have to sort m edges of graph G which takes time $O(m \log m)$. Sinc
  $O($
- As w
  making connected components which will be me
  connected components until all vertices belong t
  component. For this purpose we use Union-Find
  track the connected components constructed t
- For each edge $e = (v, u)$ on the sorted list of edges we use two FIND operations, FIND(u) and FIND(v) to determine if vertices u and v belong to the same component.

Efficient implementation of the Kruskal Algorithm

- We now use the previously described Union-Find data structure to efficiently implement the Kruskal algorithm on a graph $G = (V, E)$ with n vertices and m edges.
- We first have to sort m edges of graph G which takes time $O(m \log m)$. Sinc
  $O($
- As w
  making connected components which will be me
  connected components until all vertices belong t
  component. For this purpose we use Union-Find
  track the connected components constructed t
- For each edge $e = (v, u)$ on the sorted list of edges we use two FIND operations, FIND(u) and FIND(v) to determine if vertices u and v belong to the same component.

Efficient implementation of the Kruskal Algorithm

- If they do not belong to the same component, i.e., if $\text{FIND}(u) = i$ and $\text{FIND}(v) = j$, $j \neq i$, we add edge $e = (u, v)$ to the spanning tree being constructed and perform $\text{UNION}(i, j)$ to place u and v into the same con
- In to cost
- We also perform $n - 1$ UNION operatio
- Initial sorting of edges takes $O(m \log m$ operations; thus, we obtain an overall time compl

Efficient implementation of the Kruskal Algorithm

- If they do not belong to the same component, i.e., if $\text{FIND}(u) = i$ and $\text{FIND}(v) = j$, $j \neq i$, we add edge $e = (u, v)$ to the spanning tree being constructed and perform $\text{UNION}(i, j)$ to place u and v into the same con

- In tot n total cost

- We also perform $n - 1$ UNION operatio $\log n)$.
- Initial sorting of edges takes $O(m \log m$ g $n)$ operations; thus, we obtain an overall time compl $n)$.

Efficient implementation of the Kruskal Algorithm

- If they do not belong to the same component, i.e., if $\text{FIND}(u) = i$ and $\text{FIND}(v) = j$, $j \neq i$, we add edge $e = (u, v)$ to the spanning tree being constructed and perform $\text{UNION}(i, j)$ to place u and v into the same con

- In tot n total cost

- We also perform $n - 1$ UNION operatio $\log n)$.

- Initial sorting of edges takes $O(m \log m$ g $n)$ operations; thus, we obtain an overall time compl $n)$.

Efficient implementation of the Kruskal Algorithm

- If they do not belong to the same component, i.e., if $\text{FIND}(u) = i$ and $\text{FIND}(v) = j$, $j \neq i$, we add edge $e = (u, v)$ to the spanning tree being constructed and perform $\text{UNION}(i, j)$ to place u and v into the same con

- In tot n total cost
- We also perform $n - 1$ UNION operatio $\log n)$.

- Initial sorting of edges takes $O(m \log m$ g $n)$ operations; thus, we obtain an overall time compl $n)$.

More applications of the Greedy Method

k-clustering of maximum spacing

- **Instance:** A complete graph G with weighted edges representing distances between the two vertices.
- **Task:** Partition the vertices of G into k disjoint subsets so that the minimal dist
larg h are as
far a
- **Solution:** Sort the edges in an increasing order and start performing the usual Kruskal's algorithm for building a minimal spannin
obtain k connected components, rather than a sin
- **Proof of optimality:** Let d be the distance a
the minimal spanning tree which was not added to our k connected components; it is clearly the minimal distance between two vertices belonging to two of our k connected. Clearly, all the edges included in k many connected components produced by our algorithm are of length smaller or equal to d.

More applications of the Greedy Method

k-clustering of maximum spacing

- **Instance:** A complete graph G with weighted edges representing distances between the two vertices.
- **Task:** Partition the vertices of G into k disjoint subsets so that the minimal dist
larg h are as
far ap
- **Solution:** Sort the edges in an increasing order and start performing the usual Kruskal's algorithm for building a minimal spannin
obtain k connected components, rather than a sin
- **Proof of optimality:** Let d be the distance a
the minimal spanning tree which was not added to our k connected components; it is clearly the minimal distance between two vertices belonging to two of our k connected. Clearly, all the edges included in k many connected components produced by our algorithm are of length smaller or equal to d.

More applications of the Greedy Method

k-clustering of maximum spacing

- **Instance:** A complete graph G with weighted edges representing distances between the two vertices.
- **Task:** Partition the vertices of G into k disjoint subsets so that the minimal dist
larg h are as
far ap
- **Solution:** Sort the edges in an increasing order and start performing the usual Kruskal's algorithm for building a minimal spannin
obtain k connected components, rather than a sin
- **Proof of optimality:** Let d be the distance a
the minimal spanning tree which was not added to our k connected components; it is clearly the minimal distance between two vertices belonging to two of our k connected. Clearly, all the edges included in k many connected components produced by our algorithm are of length smaller or equal to d.

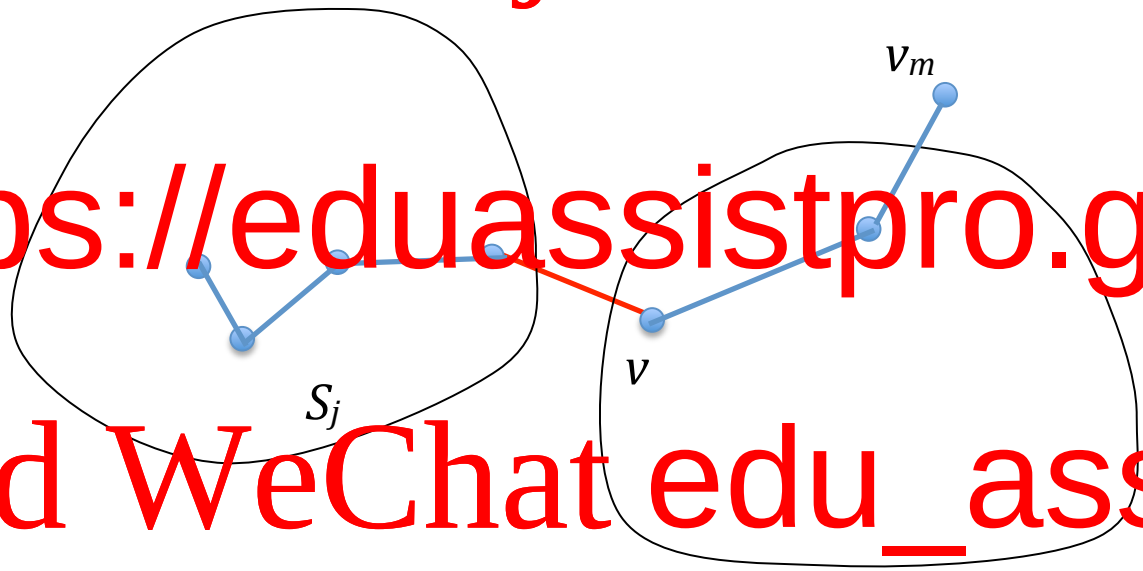
More applications of the Greedy Method

k-clustering of maximum spacing

- **Instance:** A complete graph G with weighted edges representing distances between the two vertices.
- **Task:** Partition the vertices of G into k disjoint subsets so that the minimal dist
larg h are as
far ap
- **Solution:** Sort the edges in an increasing order and start performing the usual Kruskal's algorithm for building a minimal spannin
obtain k connected components, rather than a sin
- **Proof of optimality:** Let d be the distance a
the minimal spanning tree which was not added to our k connected components; it is clearly the minimal distance between two vertices belonging to two of our k connected. Clearly, all the edges included in k many connected components produced by our algorithm are of length smaller or equal to d.

k-clustering of maximum spacing

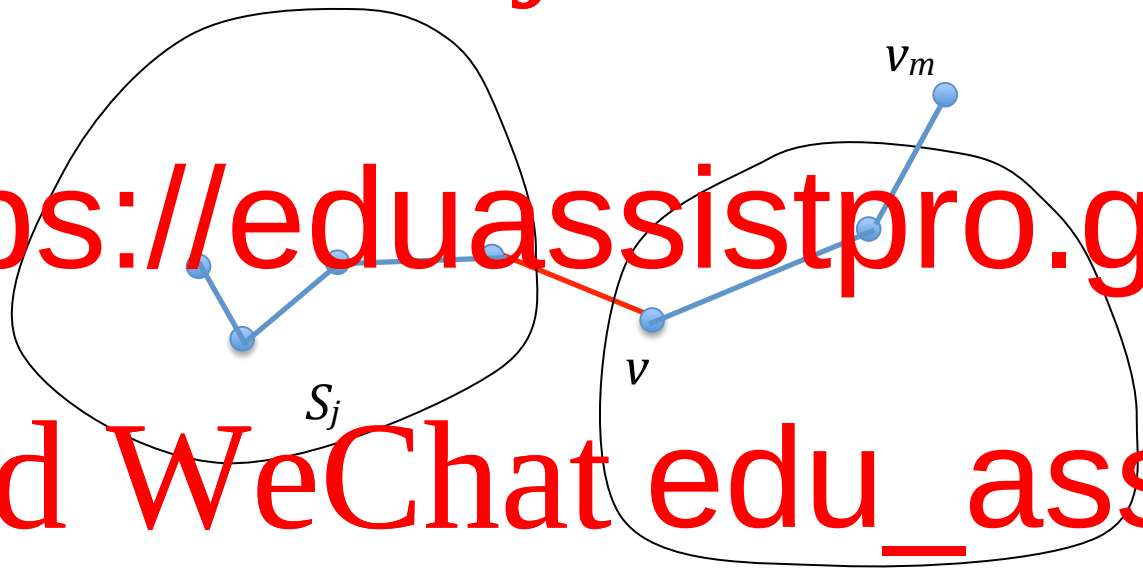
- Consider any partition $\mathcal{S}$ into k subsets different from the one produced by our algorithm.
- This means that there is a connected component produced by our algorithm which contains vertex v_i and v_m such that $v_i \in S_j$ for some $S_j \in \mathcal{S}$ and $v_m \notin S_j$.



- Since v_i and v_m belong to the same connected component, there is a path in that component connecting v_i and v_m.
- Let v_p and v_{p+1} be two consecutive vertices on that path such that v_p belongs to S_j and $v_{p+1} \notin S_j$.
- Thus, $v_{p+1} \in S_q$ for some $q \neq j$.

k-clustering of maximum spacing

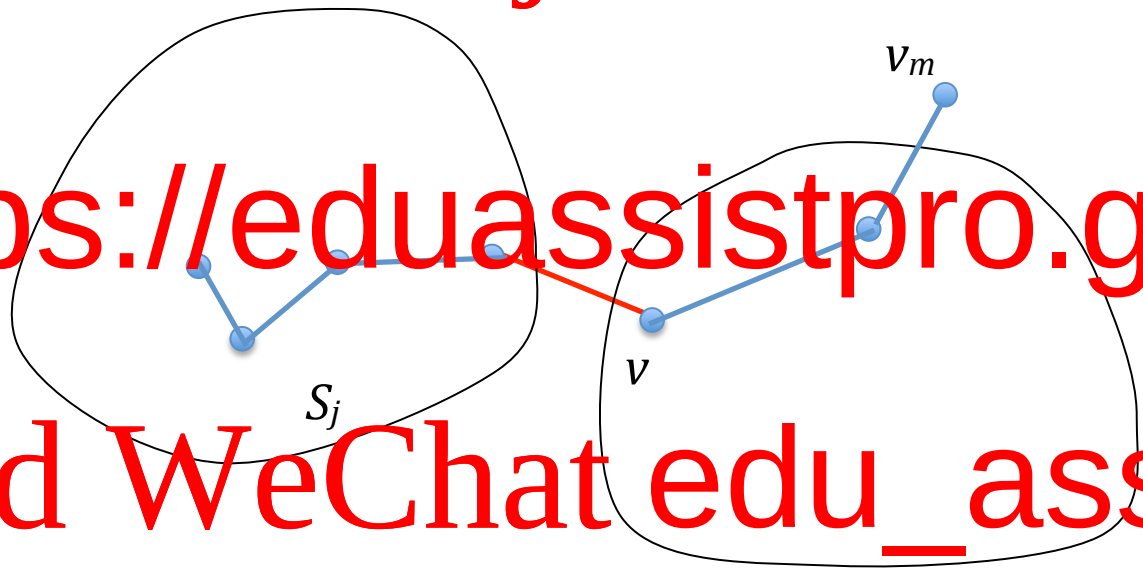
- Consider any partition $\mathcal{S}$ into k subsets different from the one produced by our algorithm.
- This means that there is a connected component produced by our algorithm which contains vertices v_i and v_m such that $v_i \in S_j$ for some $S_j \in \mathcal{S}$ and $v_m \notin S_j$.



- Since v_i and v_m belong to the same connected component, there is a path in that component connecting v_i and v_m.
- Let v_p and v_{p+1} be two consecutive vertices on that path such that v_p belongs to S_j and $v_{p+1} \notin S_j$.
- Thus, $v_{p+1} \in S_q$ for some $q \neq j$.

k-clustering of maximum spacing

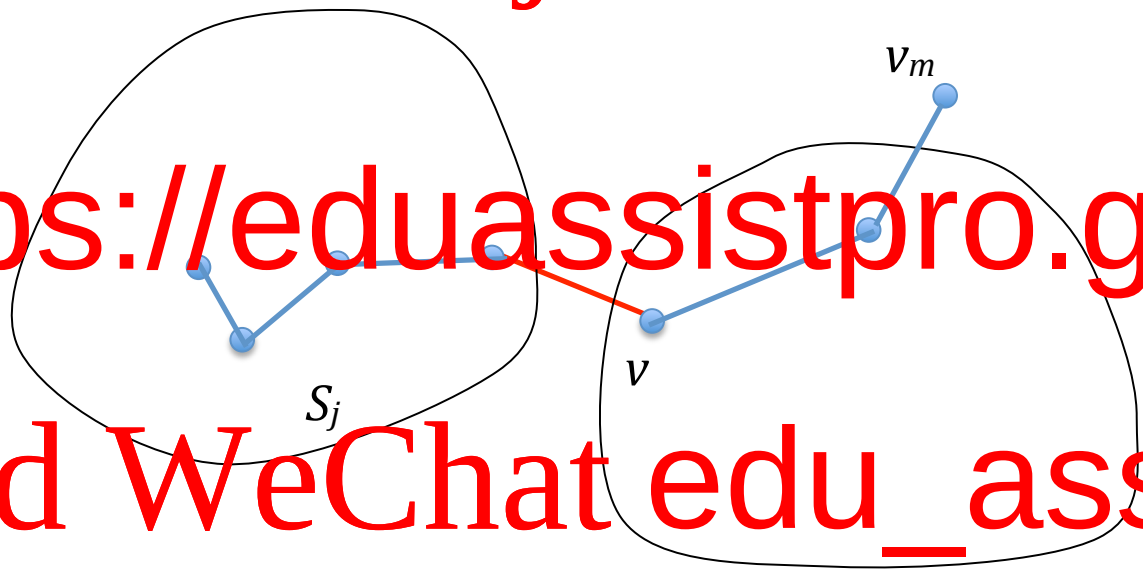
- Consider any partition $\mathcal{S}$ into k subsets different from the one produced by our algorithm.
- This means that there is a connected component produced by our algorithm which contains vertices v_i and v_m such that $v_i \in S_j$ for some $S_j \in \mathcal{S}$ and $v_m \notin S_j$.



- Since v_i and v_m belong to the same connected component, there is a path in that component connecting v_i and v_m.
- Let v_p and v_{p+1} be two consecutive vertices on that path such that v_p belongs to S_j and $v_{p+1} \notin S_j$.
- Thus, $v_{p+1} \in S_q$ for some $q \neq j$.

k-clustering of maximum spacing

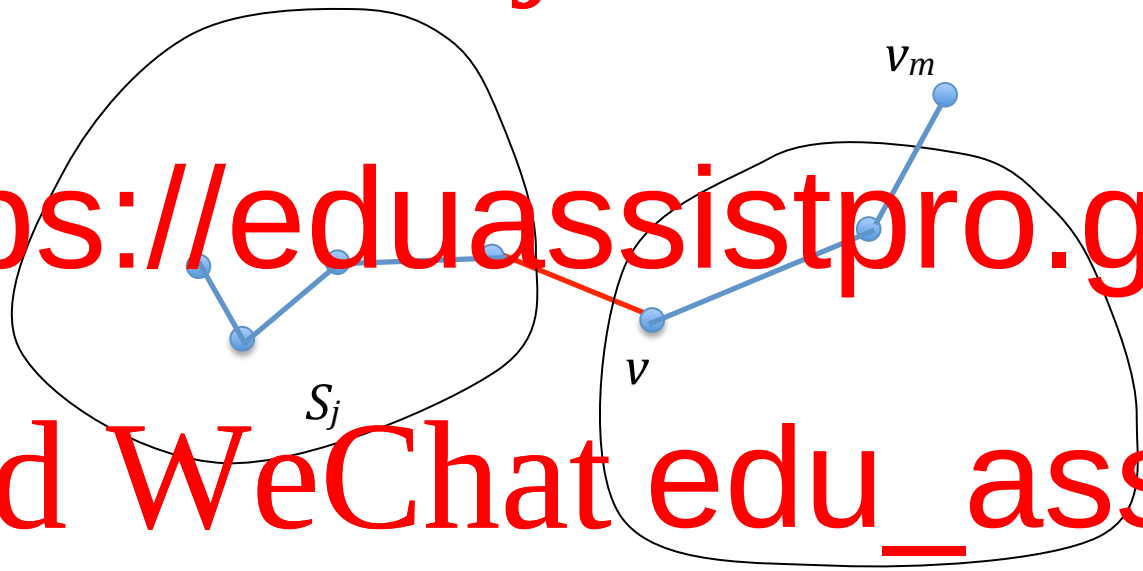
- Consider any partition $\mathcal{S}$ into k subsets different from the one produced by our algorithm.
- This means that there is a connected component produced by our algorithm which contains vertices v_i and v_m such that $v_i \in S_j$ for some $S_j \in \mathcal{S}$ and $v_m \notin S_j$.



- Since v_i and v_m belong to the same connected component, there is a path in that component connecting v_i and v_m.
- Let v_p and v_{p+1} be two consecutive vertices on that path such that v_p belongs to S_j and $v_{p+1} \notin S_j$.
- Thus, $v_{p+1} \in S_q$ for some $q \neq j$.

k-clustering of maximum spacing

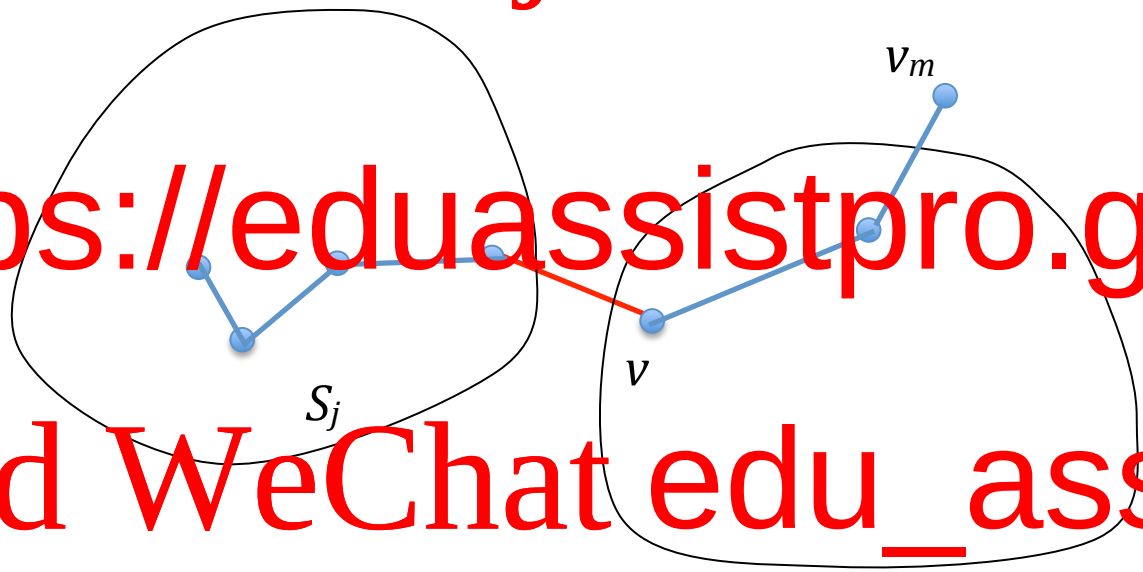
- Consider any partition $\mathcal{S}$ into k subsets different from the one produced by our algorithm.
- This means that there is a connected component produced by our algorithm which contains vertices v_i and v_m such that $v_i \in S_j$ for some $S_j \in \mathcal{S}$ and $v_m \notin S_j$.



- Since v_i and v_m belong to the same connected component, there is a path in that component connecting v_i and v_m.
- Let v_p and v_{p+1} be two consecutive vertices on that path such that v_p belongs to S_j and $v_{p+1} \notin S_j$.
- Thus, $v_{p+1} \in S_q$ for some $q \neq j$.

k-clustering of maximum spacing

- Consider any partition $\mathcal{S}$ into k subsets different from the one produced by our algorithm.
- This means that there is a connected component produced by our algorithm which contains vertices v_i and v_m such that $v_i \in S_j$ for some $S_j \in \mathcal{S}$ and $v_m \notin S_j$.



- Since v_i and v_m belong to the same connected component, there is a path in that component connecting v_i and v_m.
- Let v_p and v_{p+1} be two consecutive vertices on that path such that v_p belongs to S_j and $v_{p+1} \notin S_j$.
- Thus, $v_{p+1} \in S_q$ for some $q \neq j$.

The Greedy Method

- Note that $d(v_p, v_{p+1}) \leq d$ which implies that the distance between these two clusters $S_i, S_q \in \mathcal{S}$ is smaller or equal to the minimal distance d between the r connected components produced by our algorithm.
- Thus, such a partition cannot be a more optimal clustering than the one pro
- Wh
- We have $O(n^2)$ edges; thus sorting them by wei
  $O(n^2 \log n^2) = O(n^2 \log n)$
- While running the (partial) Kruskal algorithm -FIND data structure which requires $O(n^2 \log n)$ steps.
- So the grand total for the whole algorithm is $O(n^2 \log n)$ many steps.

The Greedy Method

- Note that $d(v_p, v_{p+1}) \leq d$ which implies that the distance between these two clusters $S_i, S_q \in \mathcal{S}$ is smaller or equal to the minimal distance d between the r connected components produced by our algorithm.
- Thus, such a partition cannot be a more optimal clustering than the one pro
- Wh
- We have $O(n^2)$ edges; thus sorting them by wei
  $O(n^2 \log n^2) = O(n^2 \log n)$
- While running the (partial) Kruskal algorithm -FIND data structure which requires $O(n^2 \log n)$ steps.
- So the grand total for the whole algorithm is $O(n^2 \log n)$ many steps.

The Greedy Method

- Note that $d(v_p, v_{p+1}) \leq d$ which implies that the distance between these two clusters $S_i, S_q \in \mathcal{S}$ is smaller or equal to the minimal distance d between the r connected components produced by our algorithm.
- Thus, such a partition cannot be a more optimal clustering than the one pro
- Wh
- We have $O(n^2)$ edges; thus sorting them by wei
  $O(n^2 \log n^2) = O(n^2 \log n)$.
- While running the (partial) Kruskal algorithm -FIND data structure which requires $O(n^2 \log n)$ steps.
- So the grand total for the whole algorithm is $O(n^2 \log n)$ many steps.

The Greedy Method

- Note that $d(v_p, v_{p+1}) \leq d$ which implies that the distance between these two clusters $S_i, S_q \in \mathcal{S}$ is smaller or equal to the minimal distance d between the r connected components produced by our algorithm.
- Thus, such a partition cannot be a more optimal clustering than the one pro
- Wh
- We have $O(n^2)$ edges; thus sorting them by wei
  $O(n^2 \log n^2) = O(n^2 \log n)$.
- While running the (partial) Kruskal algorithm -FIND data structure which requires $O(n^2 \log n)$ steps.
- So the grand total for the whole algorithm is $O(n^2 \log n)$ many steps.

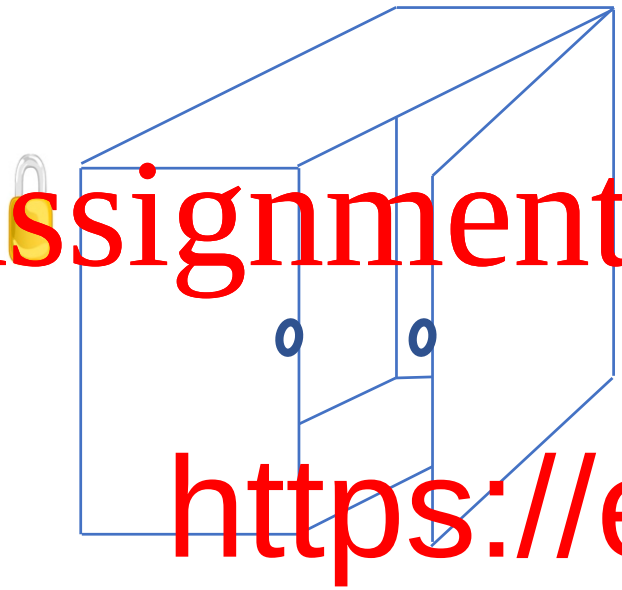
The Greedy Method

- Note that $d(v_p, v_{p+1}) \leq d$ which implies that the distance between these two clusters $S_i, S_q \in \mathcal{S}$ is smaller or equal to the minimal distance d between the r connected components produced by our algorithm.
- Thus, such a partition cannot be a more optimal clustering than the one pro
- Wh
- We have $O(n^2)$ edges; thus sorting them by wei
  $O(n^2 \log n^2) = O(n^2 \log n)$.
- While running the (partial) Kruskal algorithm -FIND data structure which requires $O(n^2 \log n)$ steps.
- So the grand total for the whole algorithm is $O(n^2 \log n)$ many steps.

The Greedy Method

- Note that $d(v_p, v_{p+1}) \leq d$ which implies that the distance between these two clusters $S_i, S_q \in \mathcal{S}$ is smaller or equal to the minimal distance d between the r connected components produced by our algorithm.
- Thus, such a partition cannot be a more optimal clustering than the one pro
- Wh
- We have $O(n^2)$ edges; thus sorting them by wei
  $O(n^2 \log n^2) = O(n^2 \log n)$.
- While running the (partial) Kruskal algorithm -FIND data structure which requires $O(n^2 \log n)$ steps.
- So the grand total for the whole algorithm is $O(n^2 \log n)$ many steps.

PUZZLE!!



The Elbonian postal service mandates that boxes to be sent, if not empty, must be locked, but they do not allow keys to be sent. The key must remain with the sender. You can send padlocks

Bob is visiting Elbonia and wishes to send his teddybear to Alice who is staying at a different hotel. Both Bob and Alice have boxes like the one shown on the picture as well as padlocks which can be used to lock the boxes. However, there is a problem.

well as that both Bob and Alice have padlocks and b
communic
on the strateg
solutions; on
solution, the other can be called the "OR" solution. The "AND" solution requires 4 mail one way services while the "OR" solution requires only 2.