# CS213/293 Data Structure and Algorithms 2024

## Lecture 16: Union-find for disjoint sets

Instructor: Ashutosh Gupta

IITB India

Compile date: 2024-10-29

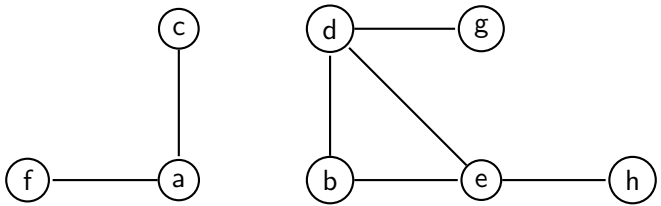
Topic 16.1

# Disjoint sets

# Example: connected components are disjoint sets

## Example 16.1

*Consider the following graph.*




The connected components are disjoint sets of nodes $\{c, a, f\}$ and $\{d, g, b, e, h\}$.

# Union-find data structure

The data structure that is used for the disjoint sets is called "union-find".

# Interface of UnionFind

UnionFind supports four interface methods

- `UnionFind<T> s` : allocates empty disjoint sets object s
- `s.makeSet(x)` : creates a new set that contains $x$.
- `s.union(x,y)` : merges the sets that contain $x$ and $y$ and returns representative of union
- `s.findSet(x)` : returns representative element of the set containing $x$.

Repeated calls to `s.findSet(x)` returns the same element if no other calls are made in between.

## Exercise 16.1

*What other methods should be in the interface?*

**Commentary:** Answer: The interface does not let you enumerate the sets. We cannot ask for the sizes. One can think of the extensions depending on the applications.

# Connected components via UnionFind

Instead of BFS, we may use UnionFind to find the connected components.

**Algorithm 16.1:** ConnectedComponents( Graph $G = (V, E)$ )

```
UnionFind s;
while v ∈ V do
    s.makeSet(v)
while {v, v'} ∈ E do
    if s.findSet(v) ≠ s.findSet(v') then
        s.union(v.v')
return s;
```

Exercise 16.2

*Given the returned s,*

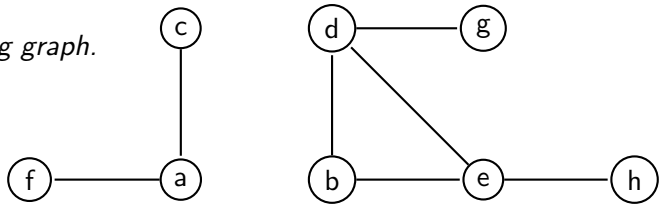
1 and 3 can be done efficiently if we, use modify the connected components function

- *can we efficiently check the number of connected components?*
- *can we efficiently check if two vertices belong to same component or not?*
- *can we efficiently enumerate the vertices of a component?*

# Example: run union-find for connected components

## Example 16.2

*Consider the following graph.*




After processing all nodes: we have sets $\{a\}$ $\{b\}$ $\{c\}$ $\{d\}$ $\{e\}$ $\{f\}$ $\{g\}$ $\{h\}$ $\{c\}$
After processing edge $\{d, b\}$: we have sets $\{a\}$ $\{b, d\}$ $\{c\}$ $\{e\}$ $\{f\}$ $\{g\}$ $\{h\}$
After processing edge $\{a, f\}$: we have sets $\{a, f\}$ $\{b, d\}$ $\{c\}$ $\{e\}$ $\{g\}$ $\{h\}$
After processing edge $\{e, h\}$: we have sets $\{a, f\}$ $\{b, d\}$ $\{c\}$ $\{e, h\}$ $\{g\}$
After processing edge $\{d, g\}$: we have sets $\{a, f\}$ $\{b, d, g\}$ $\{c\}$ $\{e, h\}$
After processing edge $\{d, g\}$: we have sets $\{a, f\}$ $\{b, d, g\}$ $\{c\}$ $\{e, h\}$
After processing edge $\{a, c\}$: we have sets $\{a, f, c\}$ $\{b, d, g\}$ $\{e, h\}$
After processing edge $\{d, e\}$: we have sets $\{a, f, c\}$ $\{b, d, g, e, h\}$
After processing edge $\{b, e\}$: we have no change.

# Recall: breadth-first based connected components

**Algorithm 16.2:** CC( Graph $G = (V, E)$ )

**for** $v \in V$ **do**
    $v.component := 0$

$componentId := 1$;
**while** *$r \in V$ such that $r.component == 0$* **do**
    BFSConnected($G, r, componentId$);
    $componentId := componentId + 1$;

In BFS, there was a step that needs to find next unvisited node after finishing a component.

Therefore, we needed to maintain a set of unvisited nodes. in the BFS method

## Exercise 16.3

*a. What is the needed interface for the set of unvisited nodes?*
*b. Can all the needed operations be done in unit cost?*

# Implementing UnionFind

If we can have efficient implementation of `findSet` and `union` then we may beat performance of BFS based connected components implementation.

What is the best can we do?

We are interested in amortised cost of the two operations, since in a practical use they are called several times.

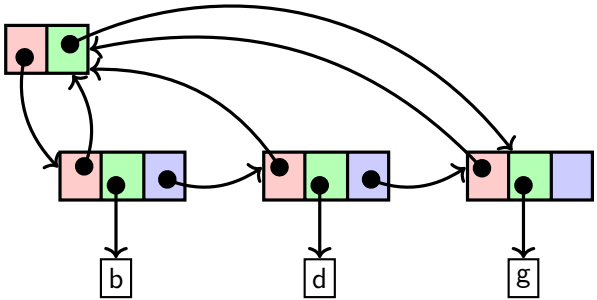
We will design an almost linear data-structure for unionFind in terms of the number of the calls to the operations.

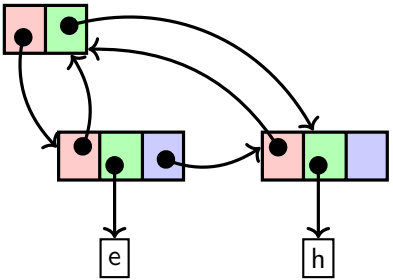
Topic 16.2

# Union-find via Linked list

# Disjoint sets representation via linked list

We need a node with two pointers to serve as the header of the set. It points to the head and tail.



# Implementing findSet and makeSet

Exercise 16.4

*a. Give implementation of makeSet?*

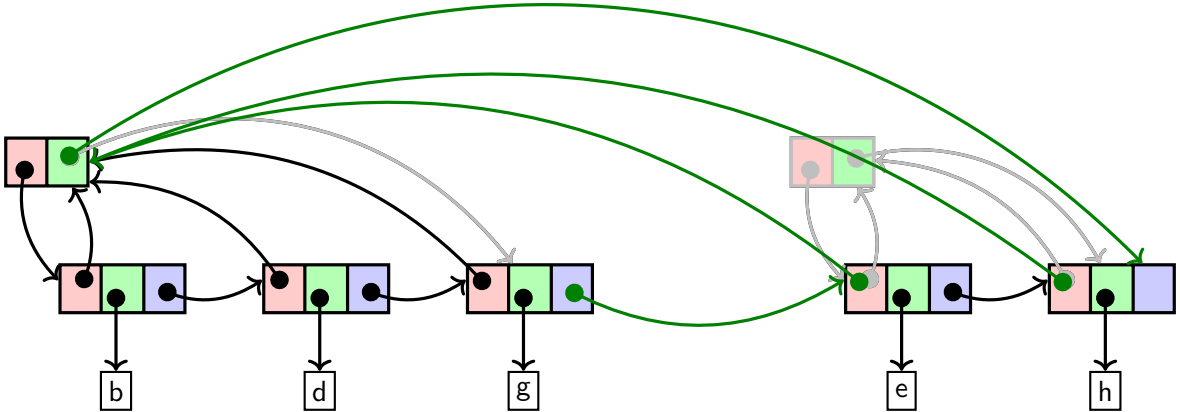
*b. Give implementation of findSet?*

**Commentary:** a. makeSet allocates a header node and a linked list, and links the nodes as suggested in the previous slide. b. findSet(node) { return node-¿header-¿head; }

# Implementing union for UnionFind

## Exercise 16.5

*On calling union(node(d), node(e)), the gray edges are removed and green edges are added.*




## Exercise 16.6

*Write code for the above transformation.*

# Running time of union

**Theorem 16.1**

*For $n$ elements, there is a sequence of $n$ union calls such that total time of the calls is $O(n^2)$.*

**Proof.**

Consider the following sequence of calls after creating nodes $x_1$,..., $x_n$.

$union(x_2, x_1)$, $union(x_3, x_2)$, $union(x_4, x_3)$, $\ldots$, $union(x_n, x_{n-1})$

At $i$th call, the union will update pointers towards headers in $i$ nodes.

Therefore, the total run time is $\sum_{n=1}^{n-1} O(i)$, which is $O(n^2)$. □

Can we do better?... Yes.

We will consider three ideas to make the running time better.

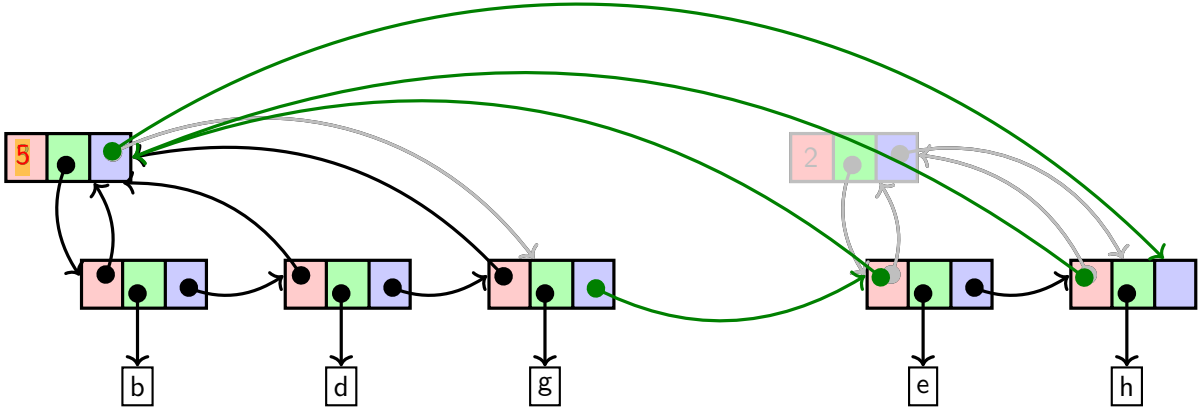
Topic 16.3

## Idea 1: weighted-union heuristics

# weighted-union heuristics

Update only the shorter list.

Add third field in the header node to store the length of the array. Exchange the parameters in $union(x, y)$ if the set containing $y$ has longer length.



# Running time with weighted-union

**Theorem 16.2**

*Each time a header pointer is updated in a node $x$, $x$ joins a set with at least double length.*

**Proof.**

Let us suppose $x$ is in a set of size $k$ and $y$ is is in a set of size $k'$.

During call $union(x, y)$, the header pointer of $x$ is updated if $k' \geq k$.

After the union, total size of the set will be $k + k' \geq 2k$. □

# Running time with weighted-union(2)

**Theorem 16.3**

*Let us suppose there are n elements in all disjoint sets. The total running time of any sequence of n unions is $O(n \log n)$*

**Proof.**

For each node the header pointer cannot be updated more than $\log n$ times.

Therefore, the total run time is $O(n \log n)$. □

Can we do better?.... **Yes**

Topic 16.4

# Idea 2: UnionFind via forest

# Each set is a tree

To avoid long traversals along the linked lists, we may represent sets via trees.

The root of tree represents the set.

Each node has only two fields: parent and size.

# Forest implementation of UnionFind

**Algorithm 16.3:** MakeSet( x )

```
x.parent = x;
x.size = 1;
```

**Algorithm 16.4:** FindSet( x )

```
if x.parent ≠ x then return FindSet( x.parent ) ;
return x.parent;
```

Exercise 16.7

*Is FindSet tail-recursive?* Yes

# Forest implementation of UnionFind(2)

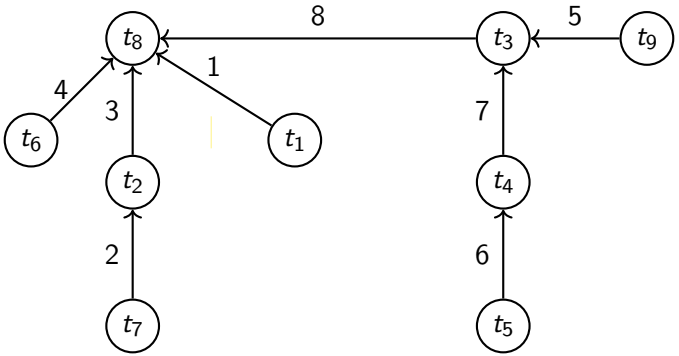
**Algorithm 16.5:** UNION( x, y )

x := FINDSET( x );
y := FINDSET( y );
**if** *x.size* < *y.size* **then** SWAP(x,y) ;
y.parent := x;
x.size = x.size + y.size

# Example: unionFind for equality reasoning via forest

## Example 16.3

*Consider:* $\underbrace{t_1 = t_8}_{1} \wedge \underbrace{t_7 = t_2}_{2} \wedge \underbrace{t_7 = t_1}_{3} \wedge \underbrace{t_6 = t_7}_{4} \wedge \underbrace{t_9 = t_3}_{5} \wedge \underbrace{t_5 = t_4}_{6} \wedge \underbrace{t_4 = t_3}_{7} \wedge \underbrace{t_7 = t_5}_{8} \wedge \underbrace{t_1 \neq t_4}_{9}$



# Running time of UnionFind via forest

**Theorem 16.4**

*The running time is still $O(n \log n)$.*

Can we do better?.... **Yes**

**Exercise 16.8**

*Show that if the height of a tree is h, then it has at least $2^{h-1}$ nodes.*

**Commentary:** Assume all trees with height $h$ have nodes $2^{h-1}$ Consider $union(x, y)$, let $y$ be the smaller tree with height $h$. Then, the number of nodes is $2^{h-1}$. The tree $T_x$ containing $x$ will have at least $2^{h-1}$ nodes. Let us suppose the height of $T_x \leq h + 1$. If the height of tree is $h + 1$ after the union, then the number nodes is more than $2^h$. If height of $T_x \geq h + 1$. There is no height change and only more nodes are added.

Topic 16.5

# Idea 3: Path compression

# Path compression

Let us directly link the a node to its representation, each time we visit a node during the run of FindSet.

**Algorithm 16.6:** FindSet( x )

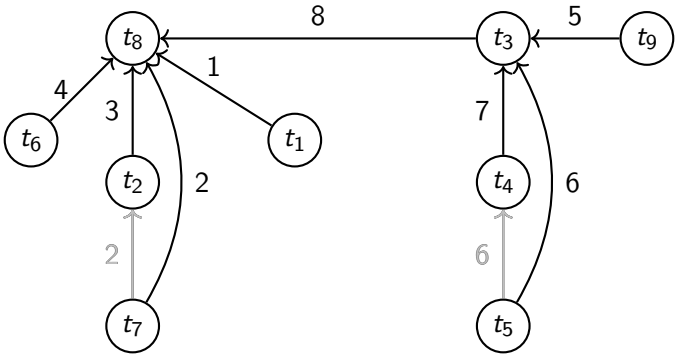
**if** $x.parent \neq x$ **then** x.parent := FindSet( x.parent ) ;
**return** x.parent;

Exercise 16.9
*Is* FindSet *tail-recursive?*

# Example: unionFind for equality reasoning with path compression

## Example 16.4

*Consider:* $\underbrace{t_1 = t_8}_{1} \wedge \underbrace{t_7 = t_2}_{2} \wedge \underbrace{t_7 = t_1}_{3} \wedge \underbrace{t_6 = t_7}_{4} \wedge \underbrace{t_9 = t_3}_{5} \wedge \underbrace{t_5 = t_4}_{6} \wedge \underbrace{t_4 = t_3}_{7} \wedge \underbrace{t_7 = t_5}_{8} \wedge \underbrace{t_1 \neq t_4}_{9}$



# Running time of UnionFind with all three ideas

The running time is $O(n\alpha(n))$, where $\alpha$ is a very slow growing function.

For any practical $n < 10^{80}$, $\alpha(n) \leq 4$.

The final data-structure is almost linear.

The proof of the above complexity is involved. Please read the text book for the proof.

**Commentary:** Watch the following interview by Tarzan who proved the bound. https://www.youtube.com/watch?v=Hhk8ANKWGJA

Topic 16.6

# Proof of work

# Proof of work

Should we be content if an algorithm says that the formula is unsatisfiable?

We must demand "why?".

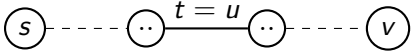
Can we generate a proof of unsatisfiablity using UnionFind data-structure?

# Proof generation in UnionFind (without path compression)

Proof generation from UnionFind data structure for an unsatisfiable input.
The proof is constructed bottom up.

1. There must be a dis-equality $s \neq v$ that was violated.
   We need to find the proof for $s = v$.

2. Find the latest edge in the path between $s$ and $v$. Let us say it is due to input literal $t = u$.




   Recursively, find the proof of $s = t$ and $u = v$.

   We stitch the proofs as follows

$$\frac{\dfrac{\dots}{s = t} \quad t = u \quad \dfrac{\dots}{u = v}}{s = v}$$

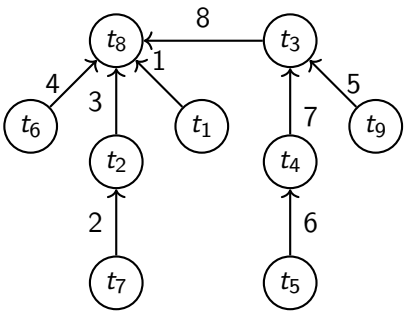
For improved algorithm: R. Nieuwenhuis and A. Oliveras. Proof-producing congruence closure. RTA'05, LNCS 3467

**Commentary:** We may need to apply symmetry rule to get the equality in right order.

# Example: union-find proof generation

## Example 16.5

*Consider:* $\underbrace{t_1 = t_8}_{1} \wedge \underbrace{t_7 = t_2}_{2} \wedge \underbrace{t_7 = t_1}_{3} \wedge \underbrace{t_6 = t_7}_{4} \wedge \underbrace{t_9 = t_3}_{5} \wedge \underbrace{t_5 = t_4}_{6} \wedge \underbrace{t_4 = t_3}_{7} \wedge \underbrace{t_7 = t_5}_{8} \wedge \underbrace{t_1 \neq t_4}_{9}$




$$\dfrac{\dfrac{\dfrac{}{t_7 = t_1}}{t_1 = t_7} \quad t_7 = t_5 \quad t_5 = t_4}{t_1 \neq t_4 \qquad t_1 = t_4}$$
$$\bot$$

1. $t_1 \neq t_4$ is violated.
2. 8 is the latest edge in the path between $t_1$ and $t_4$
3. 8 is due to $t_7 = t_5$
4. Look for proof of $t_1 = t_7$ and $t_5 = t_4$
5. 3 is the latest edge between $t_1$ and $t_7$, which is due to $t_7 = t_1$.
6. Similarly, $t_5 = t_4$ is edge 6

Topic 16.7

# Tutorial Problems

# Exercise: undo problem

Exercise 16.10

*Give an algorithm to undo the last union operation assuming there is path compression or not?*

# Exercise: tight bounds

Exercise 16.11

*Show that complexity bounds in theorem 16.3 and theorem 16.4 are tight?*

# Exercise: printSet

Exercise 16.12

*Give an implementation of printSet(x) function in UnionFind (with path compression) that prints the set containing $x$. You may add one field in each node and must not alter the asymptotic running times of the other operations.*

# End of Lecture 16

# CS213/293 Data Structure and Algorithms 2024

## Lecture 17: Graphs - minimum spanning tree

Instructor: Ashutosh Gupta

IITB India

Compile date: 2024-11-02

Topic 17.1

# Labeled graph

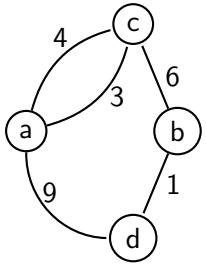
# Labeled graph

**Definition 17.1**

*A graph $G = (V, E)$ is consists of*

- *set $V$ of vertices and*
- *set $E$ of pairs of*
  - *unordered pairs from $V$ and*
  - *labels from $\mathbb{Z}^+$ (known as length/weight).*

For $e \in E$, we will write $L(e)$ to denote the length.




The above is a labeled graph $G = (V, E)$, where

$V = \{a, b, c, d\}$ and

$E = \{(\{a, c\}, 3), (\{a, c\}, 4), (\{a, d\}, 9), (\{b, c\}, 6), (\{b, d\}, 1)\}$.

$L((\{a, c\}, 3)) = 3$.

# Minimum spanning tree (MST)

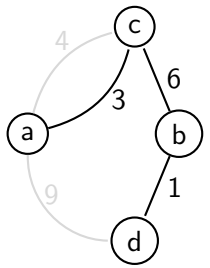
Consider a labeled graph $G = (V, E)$.

**Definition 17.2**
*A spanning tree of $G$ is a labeled tree $(V, E')$, where $E' \subseteq E$.*

**Definition 17.3**
*A length/weight of $G$ is $\sum_{e \in E} L(e)$.*

**Definition 17.4**
*A minimum spanning tree of $G$ is a spanning tree $G'$ such that the length of $G$ is minimum.*

**Example 17.1**

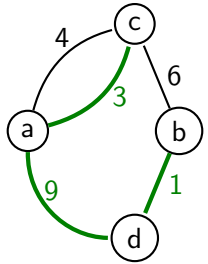



The above is an MST of the graph in the previous slide.

# Example: MST

## Example 17.2

*Consider the following spanning tree (green edges). Is this an MST?*




We can achieve an MST by replacing 9 by 6.

Observation:

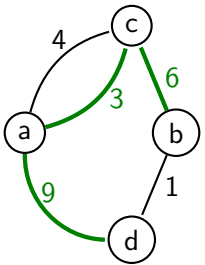
- consider an edge $e$ that is not part of the spanning tree.
- add $e$ to the spanning tree, which must form exactly one cycle.
- if $e$ is not the maximum edge in the cycle, the spanning tree is not the minimum.

# Observation: minimum edge will always be part of MST.

Apply the previous observation, the edge will replace another edge.

## Example 17.3

*In the following spanning tree, if we add edge 1, we can easily remove another edge and obtain a spanning tree with a lower length.*



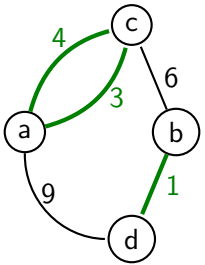

Can we keep applying this observation on greater and greater edges?

# Idea: Should we create MST using minimum $|V| - 1$ edges?

No. The method will not always work.

## Example 17.4

*In the following graph, the minimum tree edges form a cycle.*



Topic 17.2

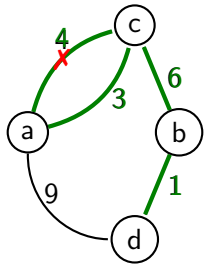
# Kruskal's algorithm

## Idea: Keep collecting minimum edges, while avoiding cycle-causing edges.

Maybe. Let us try.

### Example 17.5

*In the following graph, let us construct MST.*




It is an MST.

Are we lucky? Or, do we have a correct procedure?

# Kruskal's algorithm

**Algorithm 17.1:** MST( Graph $G = (V, E)$)

1 *Elist* := sorted list of edges in $E$ according to their labels;
2 $E' = \emptyset$;
3 **for** $e = (\{v, v'\}, \_) \in$ *Elist* **do**
4  **if** *v and v′ are not connected in* $(V, E')$ **then**
5  $E' := E' \cup \{e\}$
6 **return** $(V, E')$

In $(\{v, v'\}, \_)$, _ indicates that the value is don't care.

Running time complexity $O(\underbrace{|E|log(|E|)}_{sorting} + |E| \times IsConnected)$

# How do we check connectedness?

We maintain sets of connected vertices.

The sets merge as the algorithm proceeds.

We will show that checking connectedness can be implemented in $O(\log |V|)$ using union-find data structure.

# Example: connected sets

## Example 17.6

*Let us see connected sets in the progress of Kruskal's algorithm.*

Initial connected sets: $\{\{a\}, \{b\}, \{c\}, \{d\}\}$.

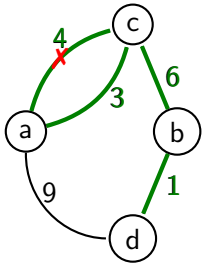
After adding edge 1: $\{\{a\}, \{c\}, \{b, d\}\}$.

After adding edge 3: $\{\{a, c\}, \{b, d\}\}$.

When we consider edge 4: $a$ and $c$ are already connected.

After adding edge 6: $\{\{a, b, c, d\}\}$.

Each time we consider an edge, we need to check if both ends of the edge are in the same set or not.



# Correctness of Kruskal's algorithm

**Theorem 17.1**
*For a graph $G = (V, E)$, $MST(G)$ returns an MST of $G$.*

**Proof.**
Let us assume edge lengths are unique. (can be relaxed!)

Suppose $MST(G)$ returns edges $e_1, ...., e_n$, which are written in increasing order.

Suppose an MST consists of edges $e'_1, ..., e'_n$, which are written in increasing order.

Let $i$ be the smallest index such that $e_i \neq e'_i$. ...

# Correctness of Kruskal's algorithm(2)

Proof(Contd.)

**case:** $L(e_i) > L(e_i')$**:**

Kruskal must have considered $e_i'$ before $e_i$.

$e_1, ...., e_{i-1}, e_i'$ has a cycle because Kruskal skipped $e_i'$.

Therefore, $e_1', ..., e_n'$ has a cycle. Contradiction. ...

# Correctness of Kruskal's algorithm(3)

## Proof(Contd.)

**case:** $L(e_i) < L(e_i')$**:**

Consider graph $e_1', ..., e_n', e_i$, which has exactly one cycle. Let $C$ be the cycle.

For all $e \in C - \{e_i\}$, $L(e_i) > L(e)$ because $e_1', ..., e_n'$ is MST. (Why?)

Therefore, $C \subseteq \{e_1, ..., e_i\}$. Beacuse all less than ei are already considered before ei so the alternate MST is already observed

Therefore, $e_1, ..., e_i$ has a cycle. Contradiction. □

b. Consider 2 MST, add an edge that is present in the 1st MST but not 2nd in the MST, now you will have a cycle, now in MST 2 also an edge would not be present in MST 1, now f belong to MST T1, not MST T2, because they are different, now either l(e) > l(f) or vice versa if 1st case remove e, then this is the MST else remove f this is the actual MST, in either case their is a unique MST

Exercise 17.1

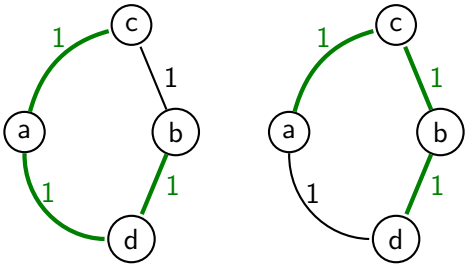
*a. Prove the above theorem after relaxing the condition of unique edge lengths.*

*b. Prove that MST is unique if edge lengths are unique.*

# Example: MST is not unique if edge lengths are not unique.

## Example 17.7

*The following graph has multiple MSTs.*



# Reverse-delete algorithm: upside down Kruskal

Here is an interesting variation of Kruskal's idea, i.e., large edges are not in MST.

**Algorithm 17.2:** MSTREVERSE( Graph $G = (V, E)$)

ASSUME($G$ is connected);
*Elist* := the list of edges in $E$ in decreasing order;
**for** $e \in$ *Elist* **do**
    **if** $(V, toSet(EList) - \{e\})$ *is a connected graph* **then**
        *EList.remove*($e$);
**return** $(V, toSet(EList))$

## Exercise 17.2

*a. Show that the algorithm returns a spanning tree.*
*b. How can we check the if condition efficiently?*

**Commentary:** Answer: a. Since the graph remains connected at the end, it will return a spanning graph. Let us suppose the algorithm returns a graph with a cycle. Consider the edge e with the maximum weight on the cycle. The algorithm must have considered e at an earlier iteration and removed it. Therefore, we have contradiction. We are yet to show that the spanning tree is minimum. b. https://dl.acm.org/doi/10.1145/335305.335345

# Correctness of MSTReverse

Let us assume all edge labels are distinct.

Theorem 17.2
$(V, toSet(EList))$ *always contains the MST.*

Proof.
**Base case:**
Initially, $G$ contains the MST.

**Induction step:**
Let us suppose at some step $(V, toSet(EList))$ contains the MST $T$ and the algorithm deletes an edge $e \in EList$.

If $e \notin T$, then $(V, toSet(EList) - \{e\})$ still contains $T$. ...

# Correctness of MSTReverse

## Proof(Contd.)

If $e \in T$, the deletion of $e$ splits $T$ into two trees $T_1$ and $T_2$.

$e$ is deleted because $(V, toSet(EList))$ has a cycle.

Therefore, there is an edge $e' \in toSet(EList) - T$, such that $T_1 \cup T_2 \cup \{e'\}$ is a tree. (Why?)

If $L(e) > L(e')$, then the weight of $T_1 \cup T_2 \cup \{e'\}$ is smaller than the weight of $T$. Contradiction.

If $L(e) < L(e')$, $e'$ must have been considered in an earlier iteration, where the same cycle was present. Therefore, $e'$ was already deleted. Contradiction. □

Topic 17.3

# Prim's algorithm

# Cut of a graph

Consider a labeled graph $G = (V, E)$.

Definition 17.5

*For a set $S \subseteq V$, a cut $C$ of $G$ is $\{(\{v, v'\}, \_) \in E | v \in S \wedge v' \notin S\}$.*

# The minimum edge of a cut will always be part of MST.

**Theorem 17.3**

*For a labeled graph $G = (V, E)$, the minimum edge of a non-empty cut $C$ will be part of MST.*

**Proof.**

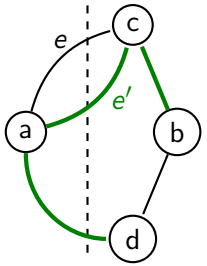
Let $C$ be a cut of $G$ for some set $S \subseteq V$ and $e \in C$ be the minimum edge.

Let us assume MST $G'$(green) does not contain $e$.

Since both $S$ and $V - S$ are not empty, $G' \cap C \neq \emptyset$ and for each $e' \in G' \cap C$ and $L(e') > L(e)$.

$G' \cup \{e\}$ has a cycle containing $e$ and some $e' \in G' \cap C$.

Therefore, $G' \cup \{e\} - \{e'\}$ is a spanning tree with a smaller length. Contradiction. □



# Prim's idea

Start with a single vertex in the visited set.

Keep expanding MST over visited vertices by adding the minimum edge connecting to the rest.

# Example: cut progress

## Example 17.8

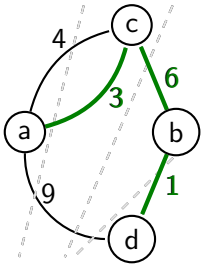
*Let us see MST construction via cuts.*

We start with vertex $a$. The cut has edges 4, 3, and 9.

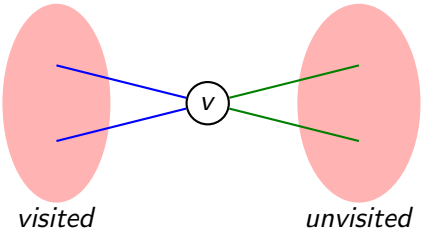
Since the minimum edge on the cut is 3, we add the edge to MST and $visited = \{a, c\}$. Now cut has edges 6 and 9.

Since the minimum edge on the cut is 6, we add the edge to MST and $visited = \{a, c, b\}$. Now cut has edges 1 and 9.

Since the minimum edge on the cut is 1, we add the edge to MST.



# Operations during entering the visited set




When a vertex $v$ moves from the unvisited set to the visited set, we need to delete blue edges from the cut and add green edges to the cut.

# Prim's algorithm

**Algorithm 17.3:** MST( Graph $G = (V, E)$, vertex $r$ )

```
1  for v ∈ V do  v.visited := False ;
2  r.visited := True;
3  for e = ({v, r}, _) ∈ E do  cut := cut ∪ {e} ;
4  while cut ≠ ∅ do
5      ({v, v'}, _) := cut.min();
6      Assume(¬v.visited ∧ v'.visited);          // This condition is always true
7      for e = ({v, w}, _) ∈ E do
8          if w.visited then
9              cut.delete(e)                     // Cost: O(log |E|)
10         else
11             cut.insert(e)                     // Cost: O(log |E|)
12     v.visited := True
```

Running time: $O(|E| \log |E|)$ because every edge will be inserted and deleted.

# Data structure for cut

We may use a heap to store the cut since we need a minimum element.

We need to be careful while deleting an edge from the heap.

Since searching in the heap is expensive, we need to keep the pointer from the edge to the node of the heap.

Can we avoid storing the set of edges? Instead, store the set of next available nodes.

# Prim's algorithm with an optimization

**Commentary:** decreasePriority reduces the priority of an element in the heap. deleteMin returns the minimum element in the heap and deletes the element in the heap. Since we are using decreasePriority for each edge, we can improve the complexity by using Fibonacci heap, where the cost of decreasePriority is amortized O(1).

**Algorithm 17.4:** MST( Graph $G = (V, E)$, vertex $r$ )

```
1  Heap unvisited;
2  for v ∈ V do
3      v.visited := False;
4      unvisited.insert(v, ∞)                          // Will heapify help?
5  unvisited.decreasePriority(r, 0);
6  while unvisited ≠ ∅ do
7      v := unvisited.deleteMin();                     // Cost: O(log |V|)
8      for e = ({v, w}, k) ∈ E do
9          if ¬w.visited then
10             unvisited.decreasePriority(w, k)        // Cost: O(log |V|)
11     v.visited := True
```

Running time: $O((|V| + |E|) \log |V|)$ because every node and edge is visited for a heap operation.

Exercise 17.3 *Modify the above algorithm to make it return the MST.*

# Example: prim with neighbors

## Example 17.9

*Let us see MST construction via unvisited neighbors.*

We start with vertex *a*. The unvisited neighbors are c and d.

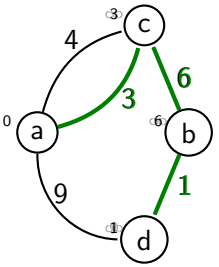
Since the minimum edge to c is 3, we update unvisited(c) $= 3$. Similarly, we update unvisited(d) $= 9$.

We add c in visited set. Now unvisited neighbors are b and d.

Since the minimum edge to b is 6, we update unvisited(b) $= 6$.

We add b in visited set. Now unvisited neighbor is d.

Since the minimum edge to d is 1, we update unvisited(d) $= 1$. We add d in visited set.



Topic 17.4

# Tutorial problems

# Exercise: proving with non-unique lengths

Exercise 17.4

*Modify proof of theorems 17.1 and 17.3 to support non-unique edges.*

# Exercise: non-unique lengths

Exercise 17.5

*Kruskal's algorithm can return different spanning trees for the same input graph G, depending on how it breaks ties when the edges are sorted into order. Show that for each minimum spanning tree T of G, there is a way to sort the edges of G in Kruskal's algorithm so that the algorithm returns T.*

# Exercise: minimum spanning tree for directed graphs

**Definition 17.6**

*A directed graph $G = (V, E)$ is a directed rooted tree(aka arborescence) if for each $v, v' \in V$ there is exactly one path between $v$ and $v'$.*

On directed graphs, the problem of finding MST changes to the problem of finding an arborescence.

**Exercise 17.6**

*a. What is anti-arborescence?*

*b. Show that Kruskal's and Prim's algorithm will not find a minimum spanning arborescence for a directed graph.*

*c. Give an algorithm that works on a directed graph.*

# Exercise: Borůvka's algorithm: Prim on steroids

Originally proposed in 1928 (before computers) to optimally layout electrical lines.

**Algorithm 17.5:** $\mathrm{MST}(\text{ Graph } G = (V, E)\ )$

Assume $G$ is connected.
mst $:= \emptyset$
components $:= \{\{v\} | v \in V\}$
**while** *components.size* $> 1$ **do**
    **for** $c \in$ *components* **do**
        e := c.outgoingEdges.min()
        mst := $mst \cup \{e\}$
    components := ConnectedComponents( (V,mst) )
**return** (V,mst)

## Exercise 17.7

*Prove that Borůvka's algorithm returns an MST.*

# End of Lecture 17

# CS213/293 Data Structure and Algorithms 2024

## Lecture 18: Graphs - Shortest path

Instructor: Ashutosh Gupta

IITB India

Compile date: 2024-11-04

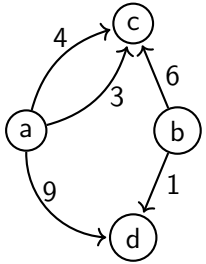
# Labeled directed graph

**Definition 18.1**

*A labeled directed graph $G = (V, E)$ is consists of*

- *set $V$ of vertices and*
- *set $E \subseteq V \times \mathbb{Q}^+ \times V$.*

For $e \in E$, we will write $L(e)$ to denote the label.




The above is a labeled graph $G = (V, E)$, where

$V = \{a, b, c, d\}$ and

$E = \{(a, 3, c), (a, 4, c), (a, 9, d), (b, 6, c), (b, 1, d)\}$.

$L((a, 3, c)) = 3$.

# Shortest path

Consider a labeled directed graph $G = (V, E)$.

## Definition 18.2

*For vertices $s, t \in V$, a path from $s$ to $t$ is a sequence of edges $e_1, ...., e_n$ from $E$ such that there is a sequence of nodes $v_1, ..., v_{n+1}$ such that $v_1 = s$, $v_{n+1} = t$, and $e_i = (v_i, -, v_{i+1})$ for each $i \in 1..n$.*

## Definition 18.3

*A length of $e_1, ..e_n$ is $\sum_{i=1}^{n} L(e_i)$.*

## Definition 18.4

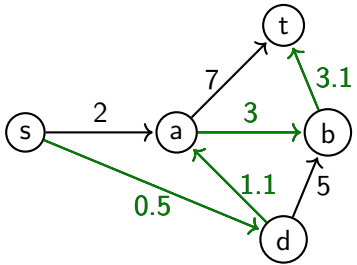
*For vertex $s, t \in V$, a shortest path is a path $s$ and $t$ such that the length of the path is minimum.*

**Commentary:** Does the above definition work for $n = 0$?

# Example: shortest path

## Example 18.1

*The shortest path from s to t is 0.5,1.1,3,3.1.*




## Exercise 18.1

*a. How many simple paths are there from s to t?*

*b. Show that there are exponentially many simple paths between two vertices.*

# Problem: single source shortest path(SSSP)

To compute a shortest path from $s$ to $t$, we need to say that there is no shorter way to reach $t$.

We need to effectively solve the following problem.

Definition 18.5

*Find shortest paths starting from a vertex $s$ to all vertices in $G$.*

Definition 18.6

*Let $SP(x)$ denote the length of a shortest path from $s$ to $x$.*

# Observation: relating SP of neighbors

An edge bounds the difference between the SP values of both ends.

For $(v, k, w) \in E$, we can conclude

$$SP(w) \leq SP(v) + k.$$

Exercise 18.2

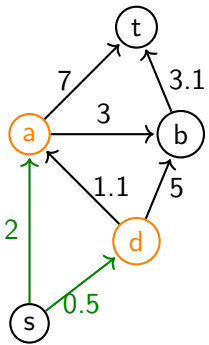
*Write a formal proof of the above claim.*

**Commentary:** Take your time to understand the above observation.

# Observation: upper bounds of paths

## Example 18.2

*Considering only outgoing edges from s, what can we say about a shortest path from s to a and d?*




$SP(s) = 0$

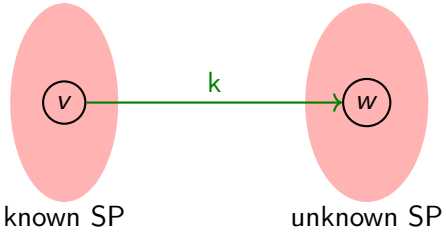
$SP(a) \leq 2 + SP(s)$

$SP(d) = 0.5 + SP(s)$

Observation: Since we know SP to $s$, we can compute SP to the closest neighbor and upper bound of SP for the other neighbors.

# Can we lift the observation for a set of nodes?

Let us suppose we know $SP$ for a set of vertices. What can we say about the remaining vertices?




$SP(w) \leq SP(v) + k$ holds for all edges that are in the cut between known and unknown.

Can we say something more about $SP(w)$ for which $SP(v) + k$ is the minimum among all edges on the cut?

# Expanding known set

Consider labeled directed graph $G = (V, E)$.

## Theorem 18.1

*Let $C$ be the cut for set $S \subset V$ in $G$.Let $d = min\{SP(v') + k | (v', k, \_) \in C\}$ and $(v, k, w) \in C$ achieves the minimum. Then, $SP(w) = d$.*

## Proof.

Let us suppose there is a path $e_1, .., e_n$ from $s$ to $w$ such that $L(e_1, ...e_n) < d$.The path has prefix

$$\underbrace{e_1 ... e_j}_{\in S} \underbrace{e_{j+1}}_{\in C} .....$$

Let $e_{j+1} = (v', k, w') \in C$. Therefore, $L(e_1, ...e_j e_{j+1}) \geq SP(v') + k$.

Due to the definition of $d$, $SP(v') + k \geq d$. Therefore, $L(e_1, .., e_n) \geq d$. Contradiction.

Therefore, $SP(w) \geq d$. □

# Dijkstra's algorithm

**Algorithm 18.1:** SSSP( Graph $G = (V, E)$, vertex $s$ )

```
1  Heap unknown; int sp[];
2  for v ∈ V do
3      v.visited := False;
4      unknown.insert(v, ∞);
5  unknown.decreasePriority(s, 0);
6  sp[s] := 0;
7  while unknown ≠ ∅ do
8      v := unknown.deleteMin();
9      for e = (v, k, w) ∈ E do
10         if ¬w.visited then
11             unknown.decreasePriority(w, k + sp[v]);
12             sp[w] := min(sp[w], k + sp[v]);
13     v.visited := True
```

# Example: Dijkstra's algorithm

Consider the following graph. We start with vertex $s$. $SP(s) = 0$.
The cut has edges 1.9 and 0.5.

The minimum path on the cut is $SP(s) + 0.5$. $SP(d) = 0.5$.

Now the cut has edges 1.9, 1, and 5.

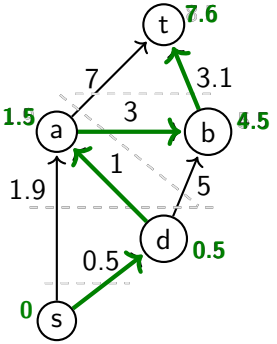
The minimum path on the cut is $SP(d) + 1$. $SP(a) = 1.5$.

Now the cut has edges 7, 3, and 5.

The minimum path on the cut is $SP(a) + 3$. $SP(b) = 4.5$.

Now the cut has edges 7 and 3.1.

The minimum path on the cut is $SP(b) + 3.1$. $SP(a) = 7.6$.



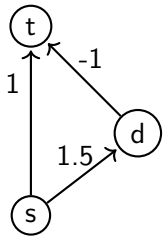

**Exercise 18.3**
*Modify Dijkstra's algorithm to construct the shortest paths from s to every vertex t.*

# Negative lengths

Dijkstra's algorithm does not work for negative lengths.

## Example 18.3

*On the following graph, Dijkstra's algorithm will return wrong shortest path.*




The algorithm uses an argument that depends on monotonic increase of length.

Topic 18.1

# A* algorithm

# Risk of exploring entire graph

Dijkstra's algorithm explores a graph without considering the target vertex.

The algorithm may explore a large part of the graph that is away from the target vertex.

A* algorithm augments Dijkstra's algorithm to be geared towards the target.

How to inform the algorithm about the potential direction of the target?

# Heuristic function $h$

$$h : V \to \mathbb{R}^+$$

$h$ estimates the shortest path from a node to the target.

## Example 18.4

*$h$ can be the Euclidian distance between two points on map.*

# A* algorithm

**Algorithm 18.2:** A*( Graph $G = (V, E)$, vertex $s$, vertex $t$, heuristic function $h$ )

```
1  Heap openSet; int sp[];
2  for v ∈ V do
3  |   sp[v] := ∞;
4  |   openSet.insert(v, ∞);
5  sp[s] := 0;
6  openSet.decreasePriority(s, sp[s] + h(s));
7  while openSet ≠ ∅ do
8  |   v := openSet.deleteMin();
9  |   if v = t then return sp[v]; ;
10 |   for e = (v, k, w) ∈ E do
11 |   |   if sp[v] + k < sp[w] then
12 |   |   |   sp[w] := sp[v] + k;
13 |   |   |   openSet.insertOrDecreasePriority(w, sp[w] + h(w));
```

# Example: A run of A*

## Example 18.5

*Consider the following four cities, which are unit distance away from the center and connected via four roads. We aim to go from $s$ to $t$.*

Let $h(v)$ be the euclidian distance of $v$ from $t$.

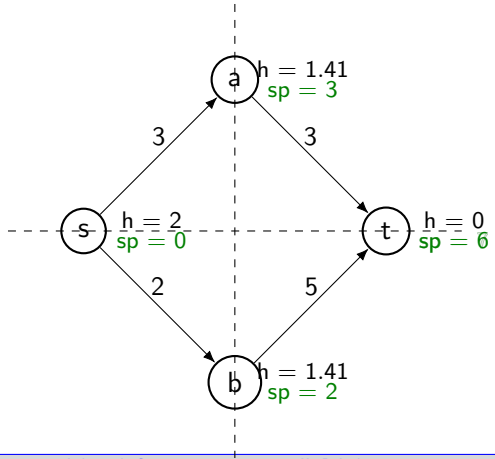
Initially: openSet $= \{s : 2\}$

After first iteration: openSet $= \{a : 4.41, b : 3.41\}$

After second iteration: openSet $= \{a : 4.41, t : 7\}$

After third iteration: openSet $= \{t : 6\}$

Last iteration: $t$ is poped and algorithm ends



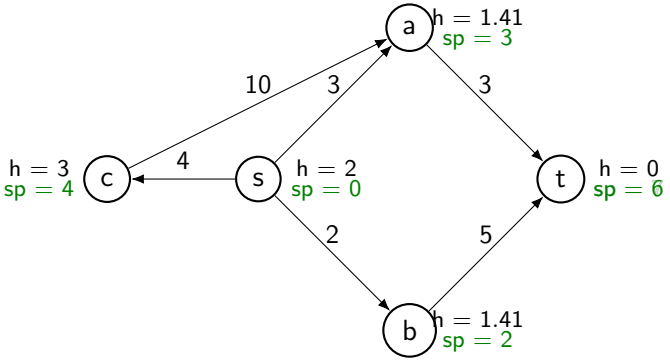
# What is the advantage of A*?

It avoids unnecessary exploration.

## Exercise 18.4

*Consider the following road network with an extra city.*

*c will enter the openSet with priority 7 and will not be popped before t.*




*h* makes $A*$ miss paths and it stops as soon as we find the target. Can we miss the shortest path?

# Correctness of A*

## Definition 18.7

*h is admissible if $h(v)$ is less than or equal to the shortest path to the target for each $v$.*

## Theorem 18.2

*If h is admissible, then A* will find the optimal path.*

## Proof.

Let us assume A* found a non-optimal path to the target $t$ with length $l'$.

Let $v_0, ....v_n$ be the shortest path to $t$ with length $l < l'$.

Let us suppose the path was not explored by A* beyond $v_i$.

$v_i$ must have entered openSet and never left. ...

# Correctness of A* (2)

Proof.
The priority of $v_i$ in openSet is $sp[v_i] + h(v_i)$.

Since $h$ is admissible, $sp[v_i] + h(v_i) \leq sp[v_i] + Length(v_{i+1}, ...., v_n) \leq l$.

Since $t$ was deleted from openSet before $v_i$, the priority of $v_i$ in OpenSet must be greater than $l'$.
Contradiction. □

# Connection with Dijkstra's algorithm

A* is more general algorithm than Dijkstra's algorithm.

Dijkstra's algorithm is unguided but never visits a node again.

In A*, a vertex can leave openSet and enter again. When A* is like Dijkstra?

## Definition 18.8

*h is consistent if for each* $(v, k, w) \in E$, $h(v) \leq h(w) + k$.

## Theorem 18.3

*If h is consistent, all vertex will enter openSet only once.*

## Exercise 18.5

*a. Show that if* $h(v) = 0$ *for all* $v$, *A* is Dijkstra's algorithm.*
*b. Prove the above theorem.*

Topic 18.2

# Tutorial problems

# Example: Counting paths

Exercise 18.6

*Modify Dijkstra's algorithm to compute the number of shortest paths from $s$ to every vertex $t$.*

# Example: Negative edges

Exercise 18.7

*Show an example of a graph with negative edge weights and show how Dijkstra's algorithm may fail. Suppose that the minimum negative edge weight is -d. Suppose that we create a new graph G' with weights w', where G' has the same edges and vertices as G, but w'(e)=w(e)+d. In other words, we have added d to every edge weight so that all edges in the new graph have edge weights non-negative. Let us run Dijkstra on this graph. Will it return the shortest paths for G?*

# Example: Road network

## Exercise 18.8

*Let G(V,E) be a representation of a geography with V as cities and (u,v) an edge if and only if there is a road between the cities u and v. Let d(u,v) be the length of this road. Suppose that there is a bus plying on these roads with fare f(u,v)=d(u,v). Next, suppose that you have a free coupon that allows you one free bus ride. Find the least fare paths from s to another city v using the coupon for this travel.*

## Exercise 18.9

*Same as above. Suppose w(u,v) is the width of the road between the cities u and w. Given a path pi, the width w(pi) is the minimum of widths of all edges in pi. Given a pair of cities s and u, is it possible to use Dijkstra to determine d such that it is the largest width of all paths pi from s to v?*

## Exercise 18.10

*Same as above. For a path pi, define hop(pi)=max(d(e)) for all e in pi. Thus if one is traveling on a motorcycle and if fuel is available only in cities, then hop(pi) determines the fuel capacity of the tank of your motorcycle needed to undertake the trip. Now, for any s and u, we want to determine the minimum of hop(pi) for all paths pi from s to u. Again, can Dijkstra be used?*

Topic 18.3

# Problems

# Example: Travel plan

Exercise 18.11

*You are given a timetable for a city. The city consists of n stops V=v1,v2,...,vn. It runs m services s1,s2,...,sm. Each service is a sequence of vertices and timings. For example, the schedule for service K7 is given below. Now, you are at stop A at 8:00 a.m. and you would like to reach stop B at the earliest possible time. Assume that buses may be delayed by at most 45 seconds. Model the above problem as a shortest path problem. The answer should be a travel plan.*

# Example: Preferred paths

Exercise 18.12

*Given a graph G(V,E) and a distinguished vertex s and a vertex v, there may be many shortest paths from s to v. What shortest path is identified by Dijkstra?*

# End of Lecture 18