
# INF421, Lecture 2
# Queues

Leo Liberti

LIX, École Polytechnique, France

# Course

- **Objective**: to teach you some data structures and associated algorithms
- **Evaluation**: TP noté en salle info le 16 septembre, Contrôle à la fin. Note: $\max(CC, \frac{3}{4}CC + \frac{1}{4}TP)$
- **Organization**: fri 26/8, 2/9, 9/9, 16/9, 23/9, 30/9, 7/10, 14/10, 21/10, amphi 1030-12 (Arago), TD 1330-1530, 1545-1745 (SI31,32,33,34)
- **Books**:
  1. Ph. Baptiste & L. Maranget, *Programmation et Algorithmique*, Ecole Polytechnique (Polycopié), 2006
  2. G. Dowek, *Les principes des langages de programmation*, Editions de l'X, 2008
  3. D. Knuth, *The Art of Computer Programming*, Addison-Wesley, 1997
  4. K. Mehlhorn & P. Sanders, *Algorithms and Data Structures*, Springer, 2008
- **Website**: `www.enseignement.polytechnique.fr/informatique/INF421`
- **Contact**: `liberti@lix.polytechnique.fr` (e-mail subject: INF421)

# Remarks on TD1

- Every object must be **initialized**

  `object` = something meaningful;

- If `L` is a list of the Java library,

  ```
  LinkedList<Integer> L = null;
  ```

  is **NOT THE EMPTY LIST**!!!!

- Try

  ```
  LinkedList<Integer> L = new LinkedList<Integer>();
  ```

  instead!

- In some implementations of a list, `null` might be the empty list, though — just pay attention to the instructions for using the implementation at hand


# Answers to your comments

1. **Rhythm**: most are happy, some find it slow. *Your class has a mixed level, it's not going to be easy to make everyone happy*. I aim to keep the rhythm **slow** but escalate the contents' difficulty
2. **Some material not new**: this is for pedagogical reasons. If you see different people presenting the same material in different ways, you'll understand it much better. Remember, this is still a foundational course
3. **Slides in print format**: OK (you are SPOILED! the slides in print format were ALREADY on the website, you could have printed them yourselves!)
4. **Code**: my slides will contain pseudocode, but I'll show you some Java code at the end of each lecture
5. **Liveliness**: I enjoy lecturing and hope you'll always find the course lively, but there will certainly be times when you get bored. I apologize in advance and hope these will be kept to a minimum


# Lecture summary

- A motivating example
- Queues
- Breadth-First Search (BFS)
- Implementation

# Motivating example

# Bus network with timetables

| A | |
|---|---|
| 1 | h:00 |
| 2 | h:10 |
| 3 | h:30 |

| B | |
|---|---|
| 1 | h:00 |
| 4 | h:20 |
| 5 | h:40 |

| C | |
|---|---|
| 2 | h:10 |
| 3 | h:20 |
| 5 | h:30 |

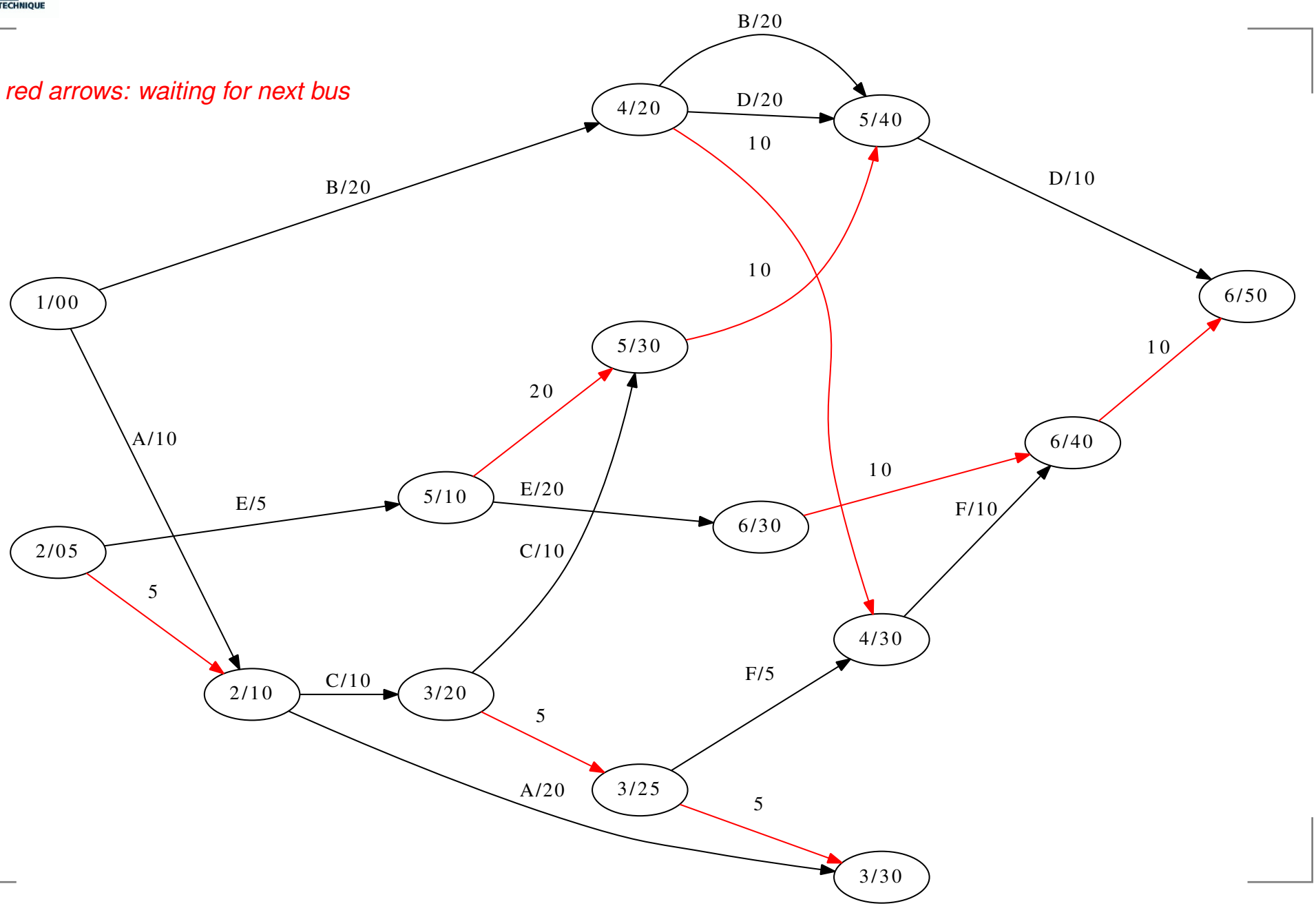
| D | |
|---|---|
| 4 | h:20 |
| 5 | h:40 |
| 6 | h:50 |

| E | |
|---|---|
| 2 | h:05 |
| 5 | h:10 |
| 6 | h:30 |

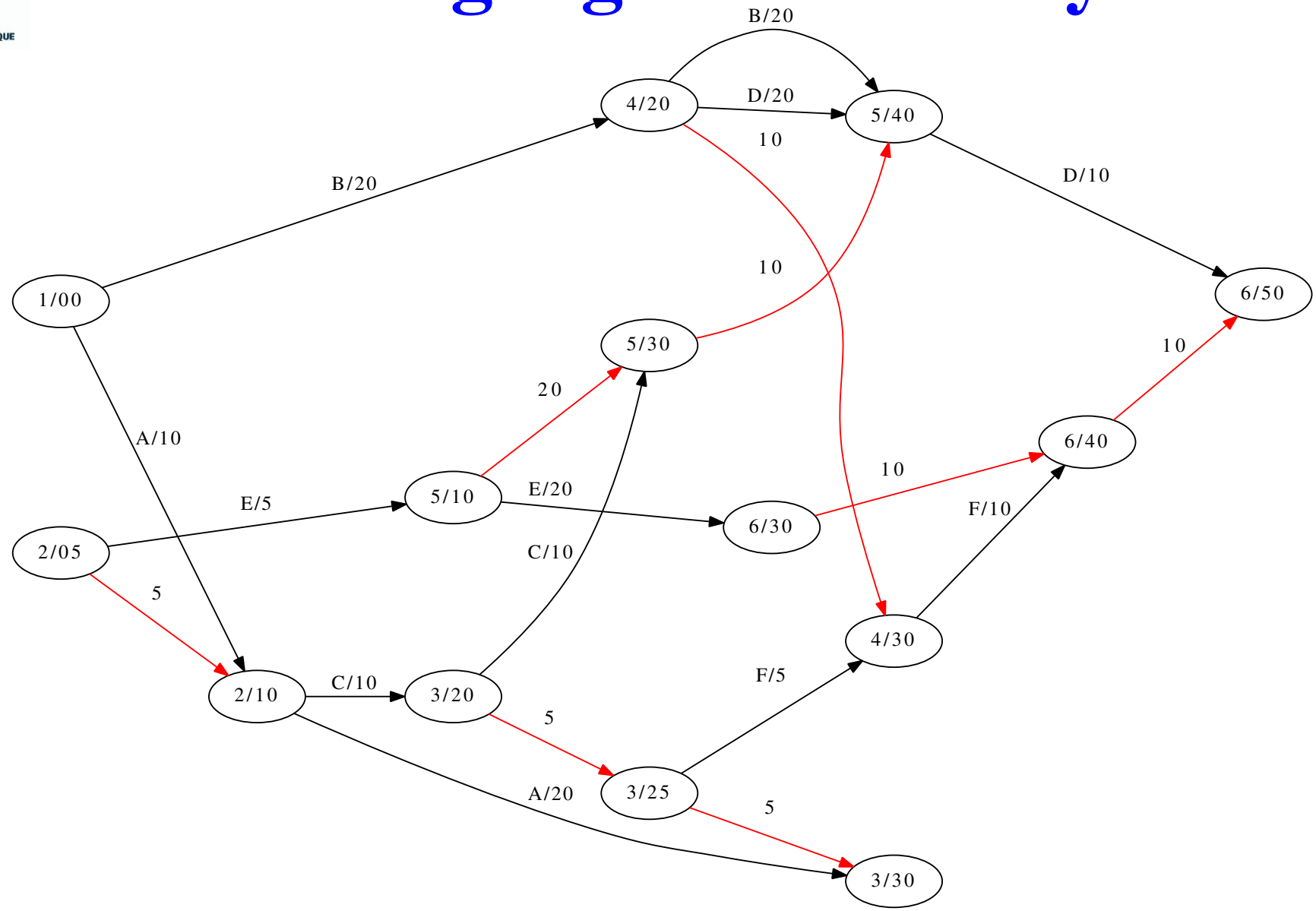
| F | |
|---|---|
| 3 | h:25 |
| 4 | h:30 |
| 6 | h:40 |

Find a convenient itinerary from 1 to 6, leaving at h:00?

# The event graph

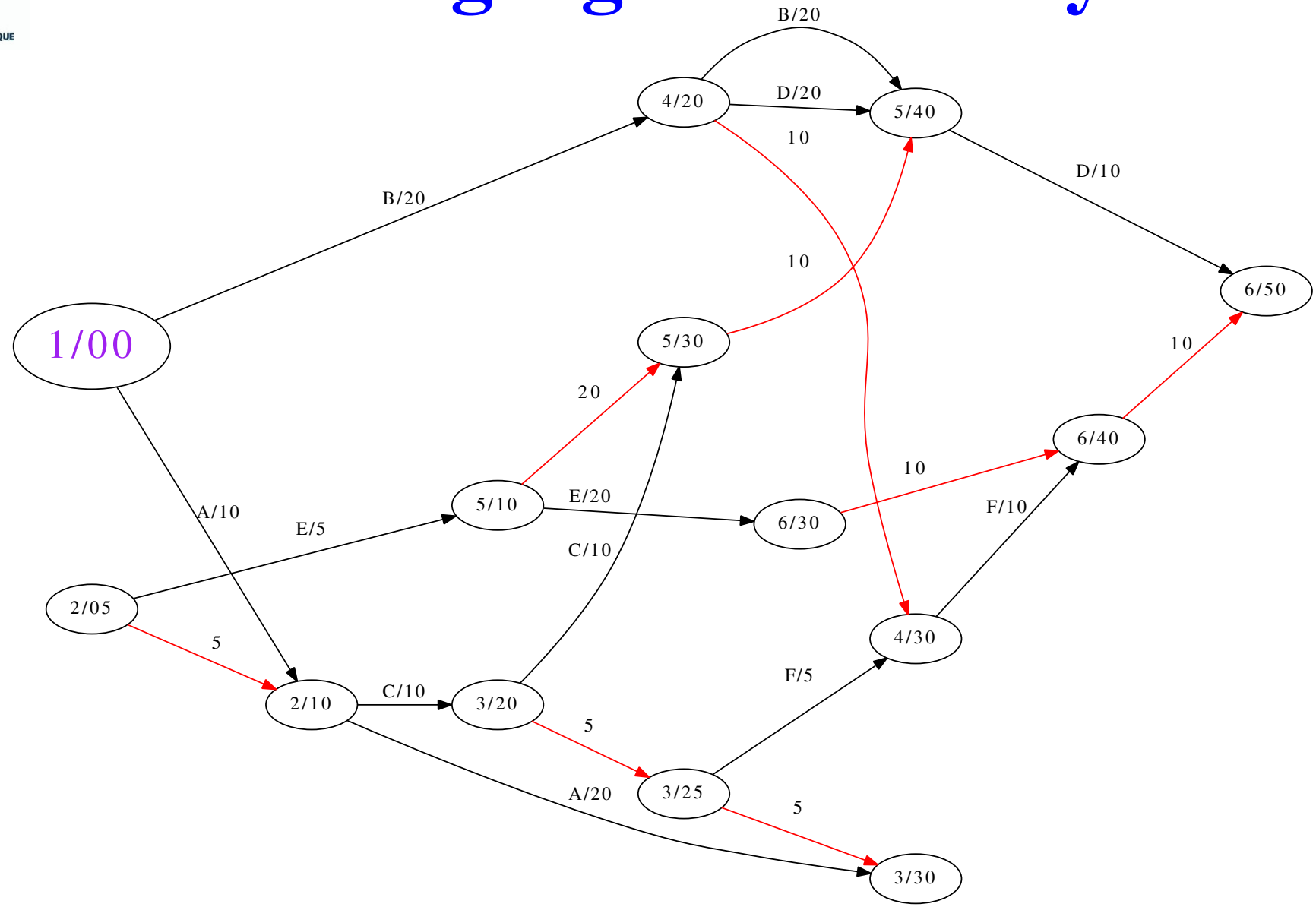


# Finding a good itinerary



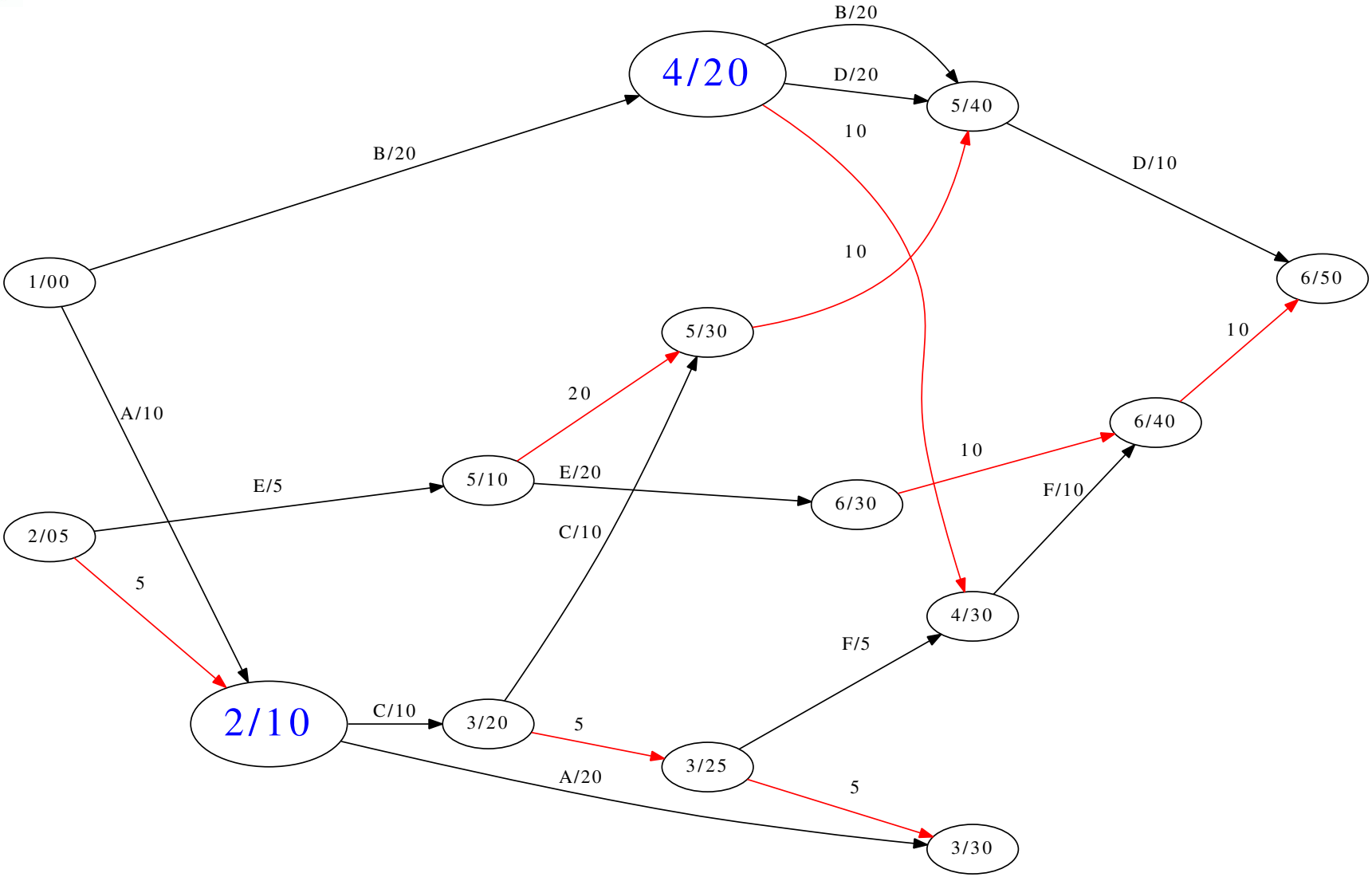

1/00

# Finding a good itinerary



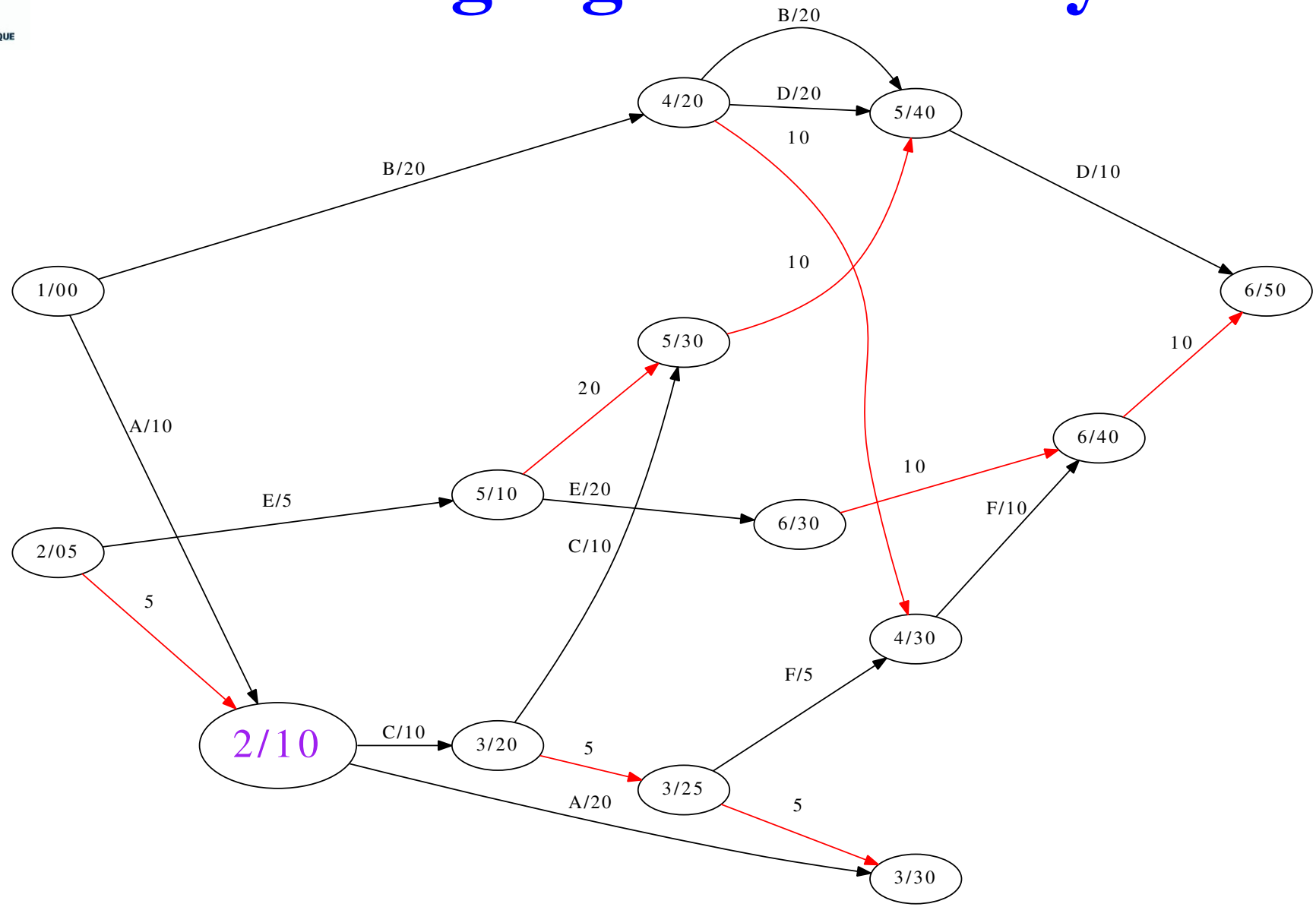

1/00←

# Finding a good itinerary



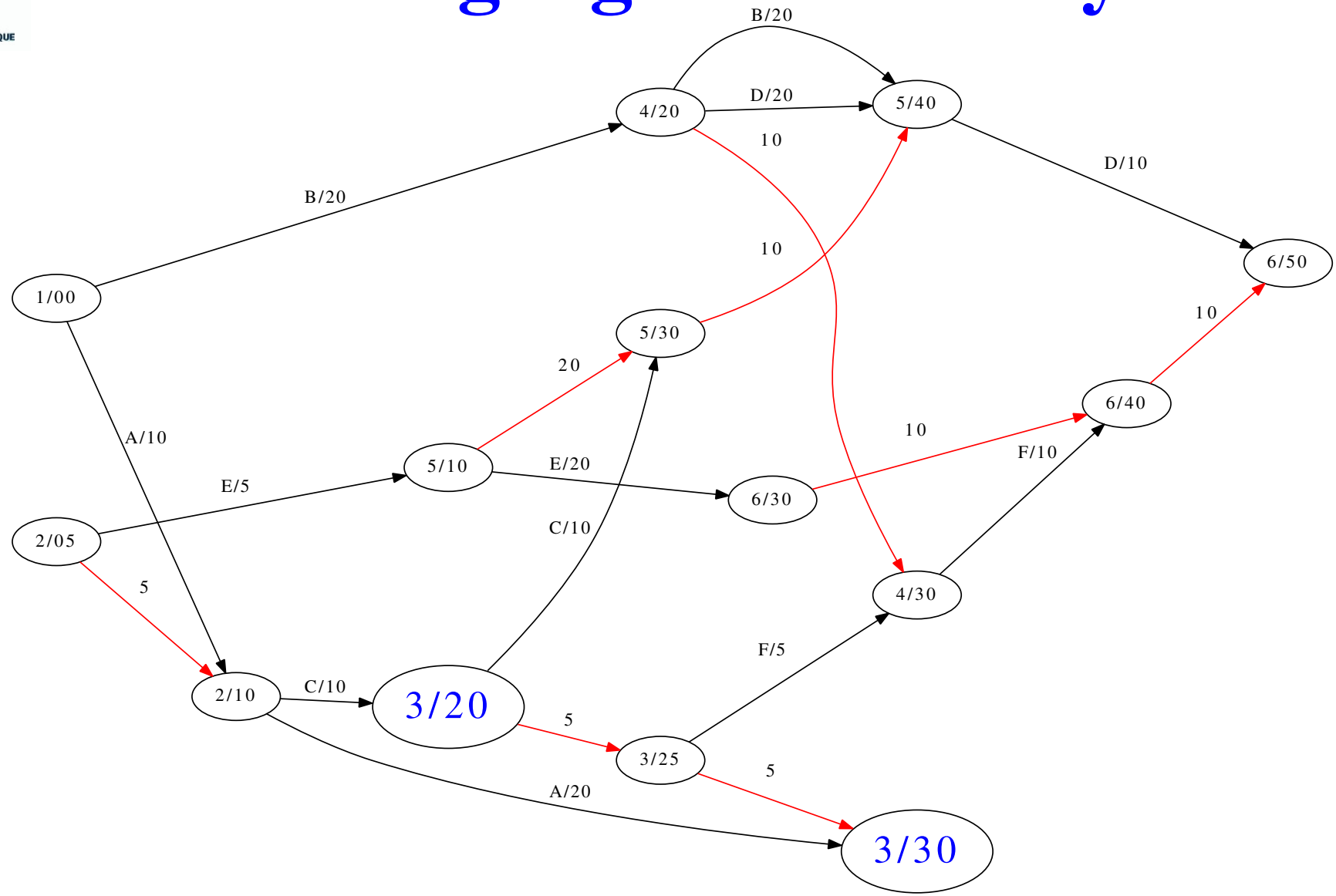

| 2/10 | 4/20 |
|---|---|

# Finding a good itinerary



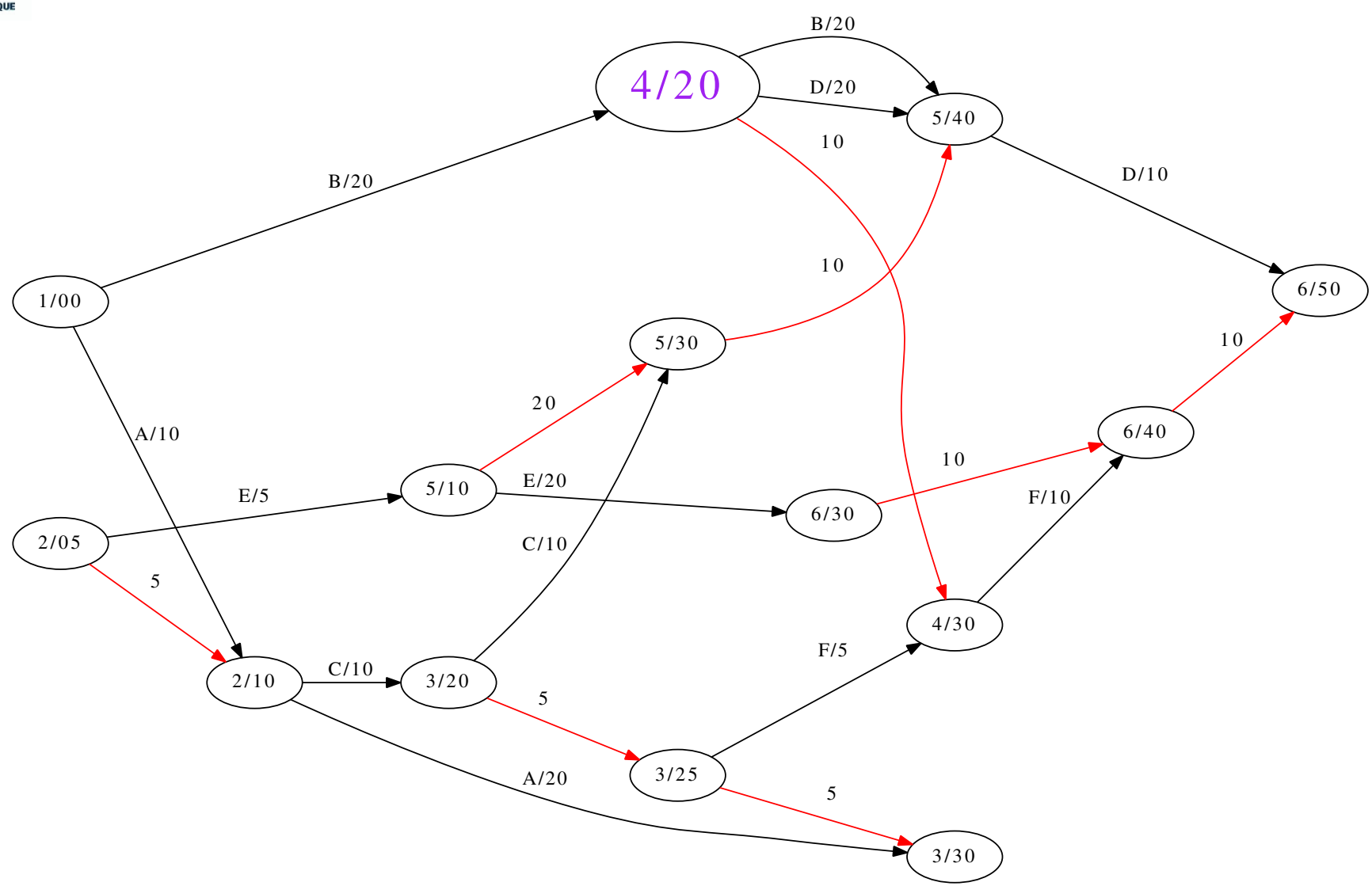

2/10← 4/20

# Finding a good itinerary



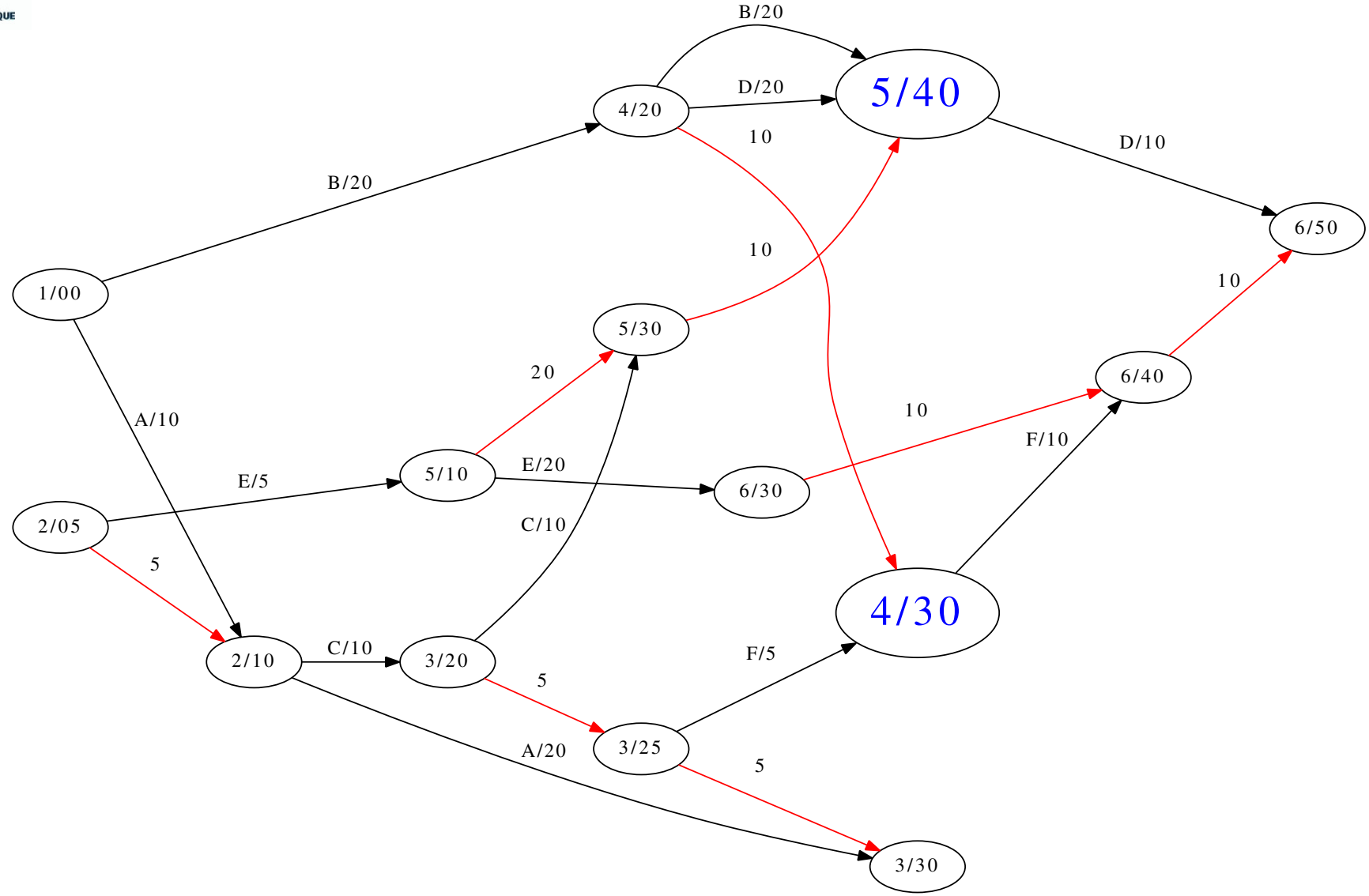

| 4/20 | 3/20 | 3/30 |
|---|---|---|

# Finding a good itinerary



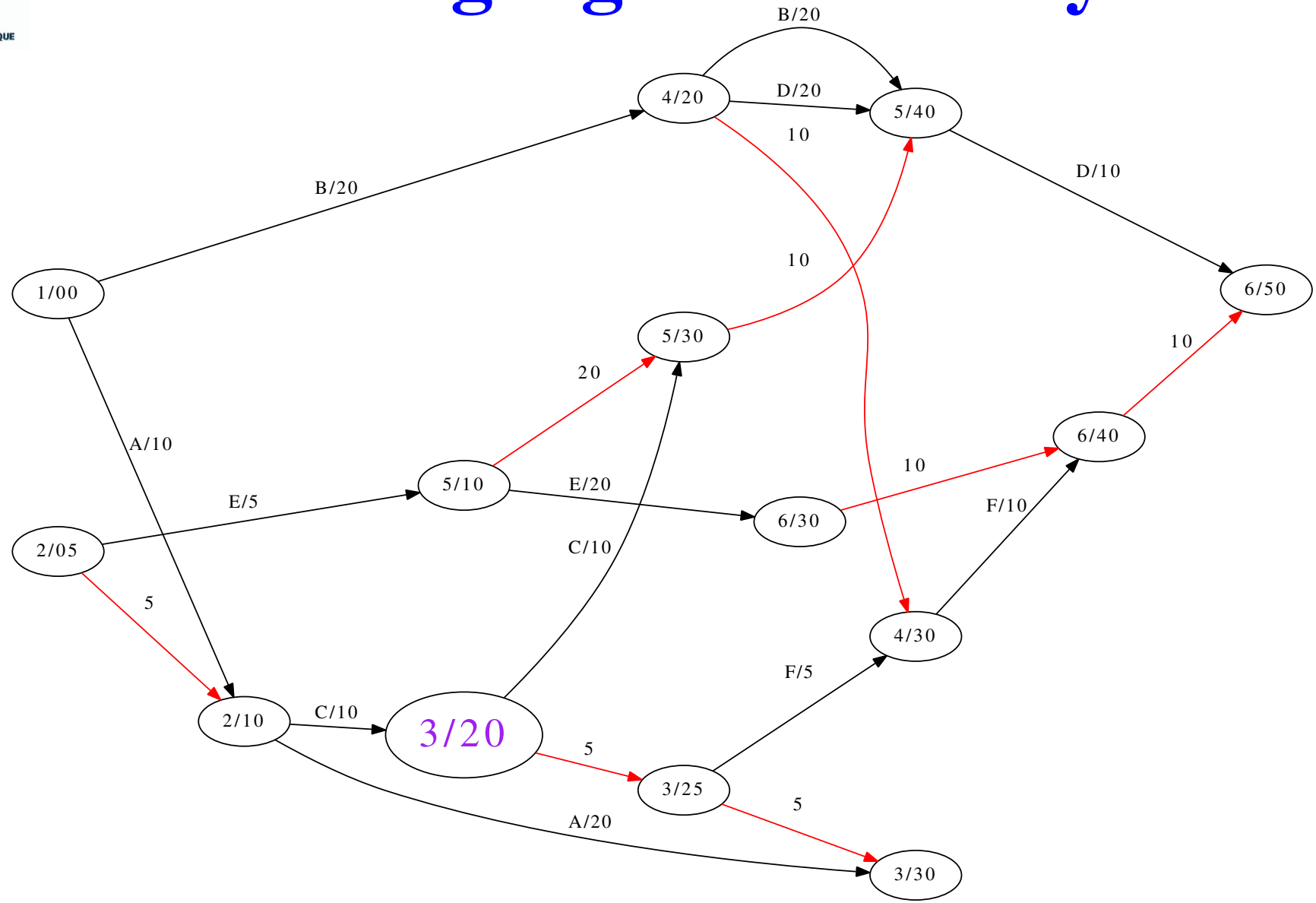

4/20← | 3/20 | 3/30 |

# Finding a good itinerary



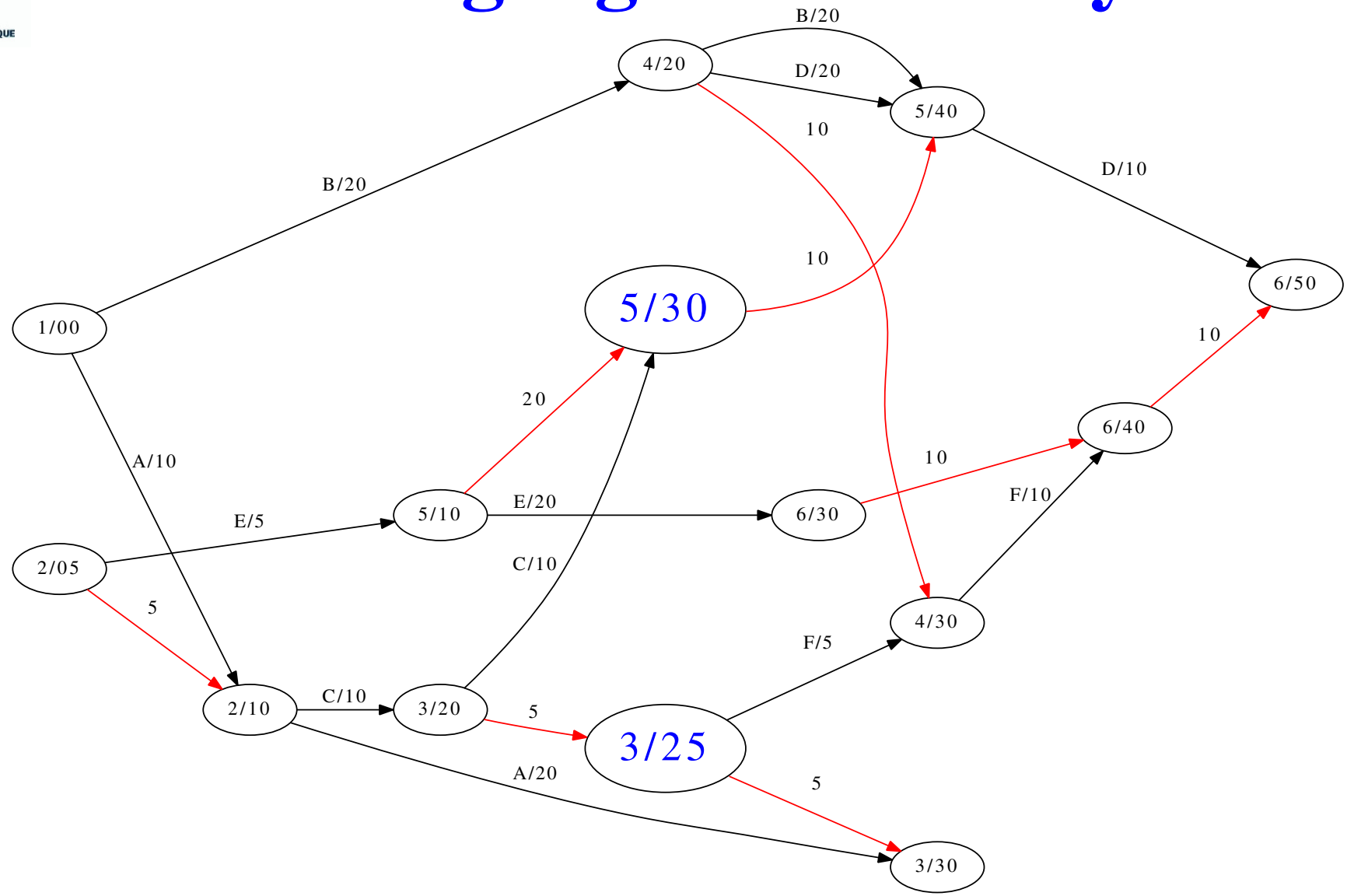

| 3/20 | 3/30 | 4/30 | 5/40 |
|---|---|---|---|

# Finding a good itinerary



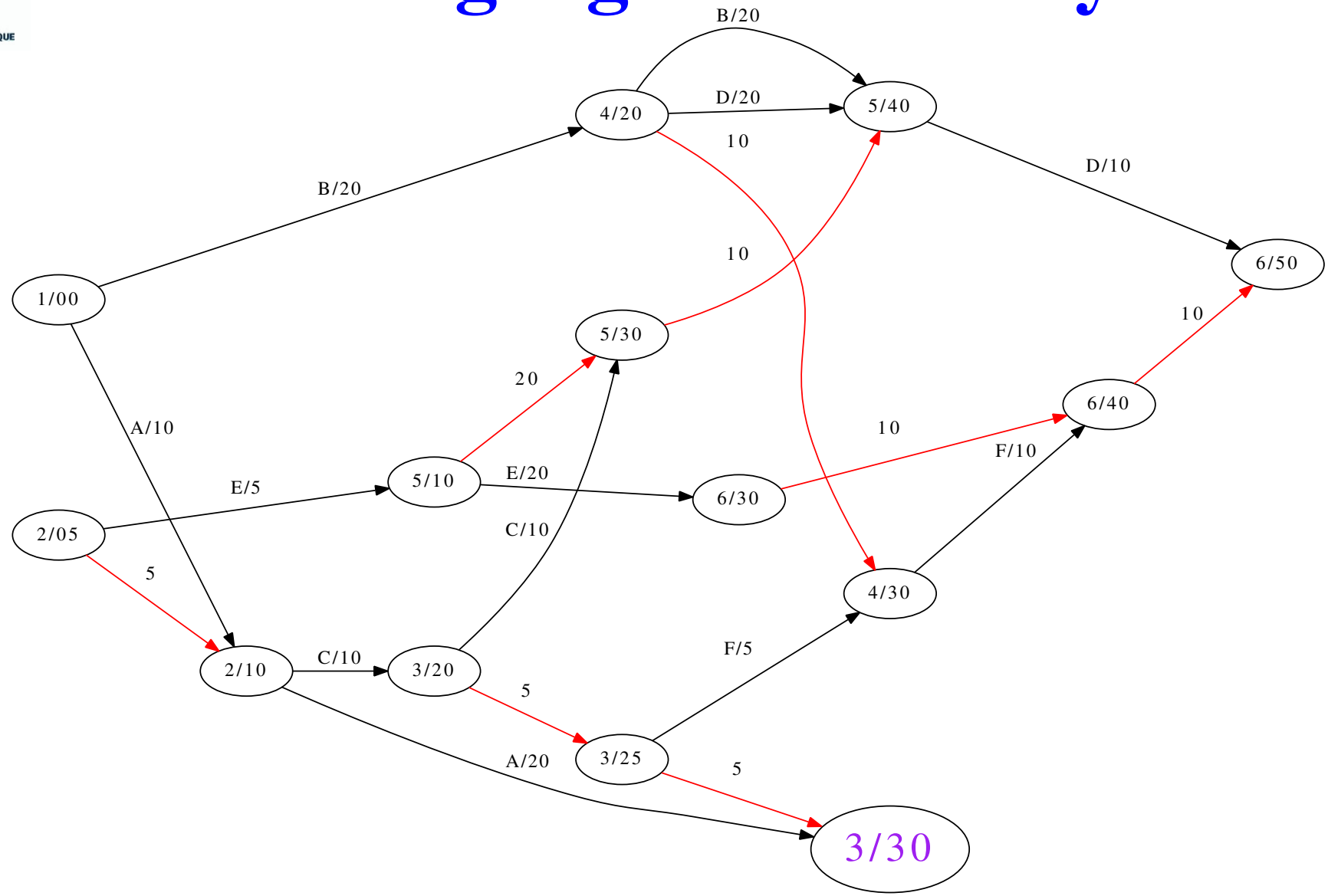

3/20← | 3/30 | 4/30 | 5/40 |

# Finding a good itinerary



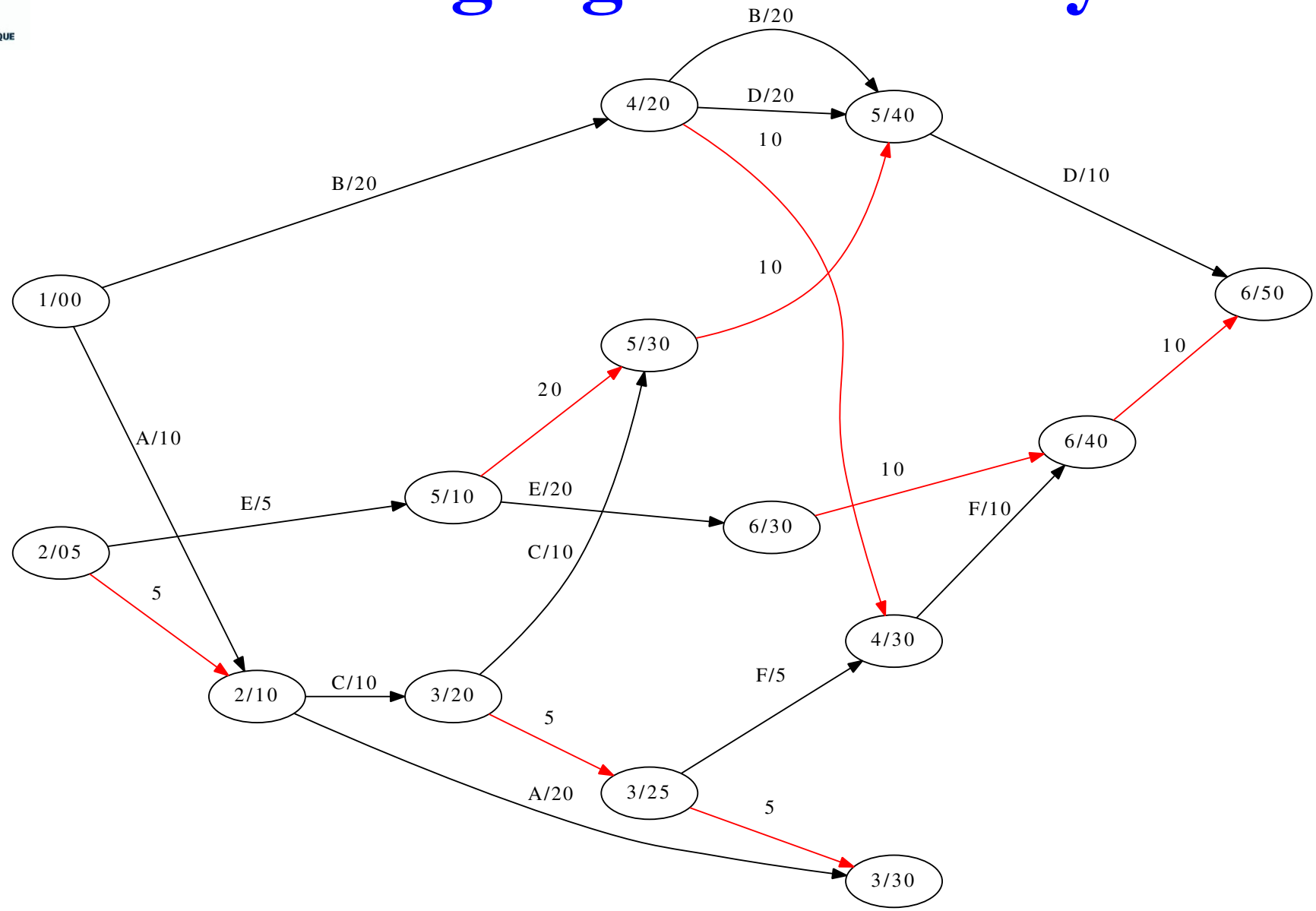

| | | | | |
|---|---|---|---|---|
| 3/30 | 4/30 | 5/40 | 3/25 | 5/30 |

# Finding a good itinerary



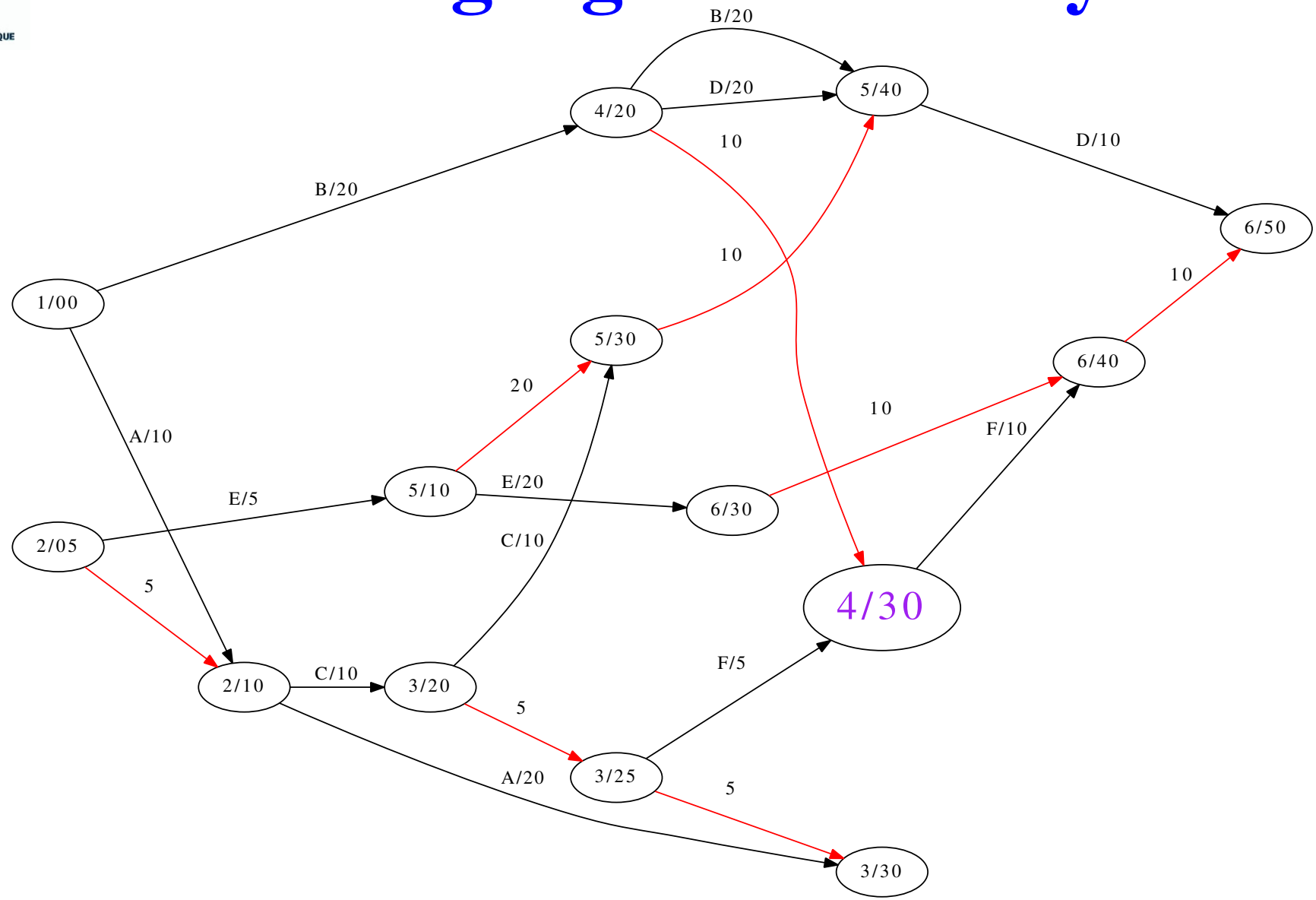

3/30← | 4/30 | 5/40 | 3/25 | 5/30 |

# Finding a good itinerary



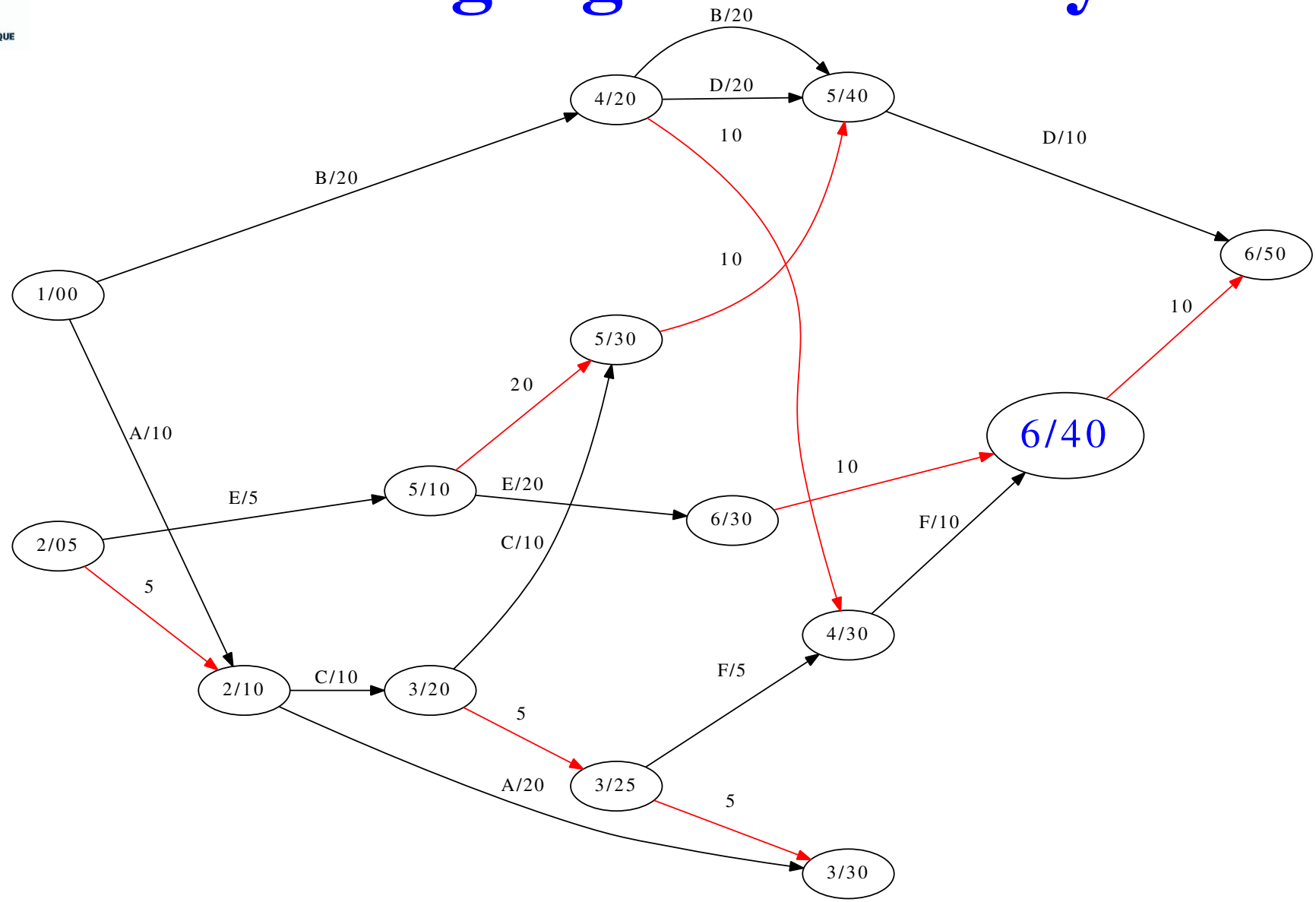

| 4/30 | 5/40 | 3/25 | 5/30 |
|---|---|---|---|

# Finding a good itinerary




4/30← | 5/40 | 3/25 | 5/30 |

# Finding a good itinerary




| 5/40 | 3/25 | 5/30 | 6/40 |
|---|---|---|---|

*found itinerary 1→6 arriving at h:40*

# Retrieving the path

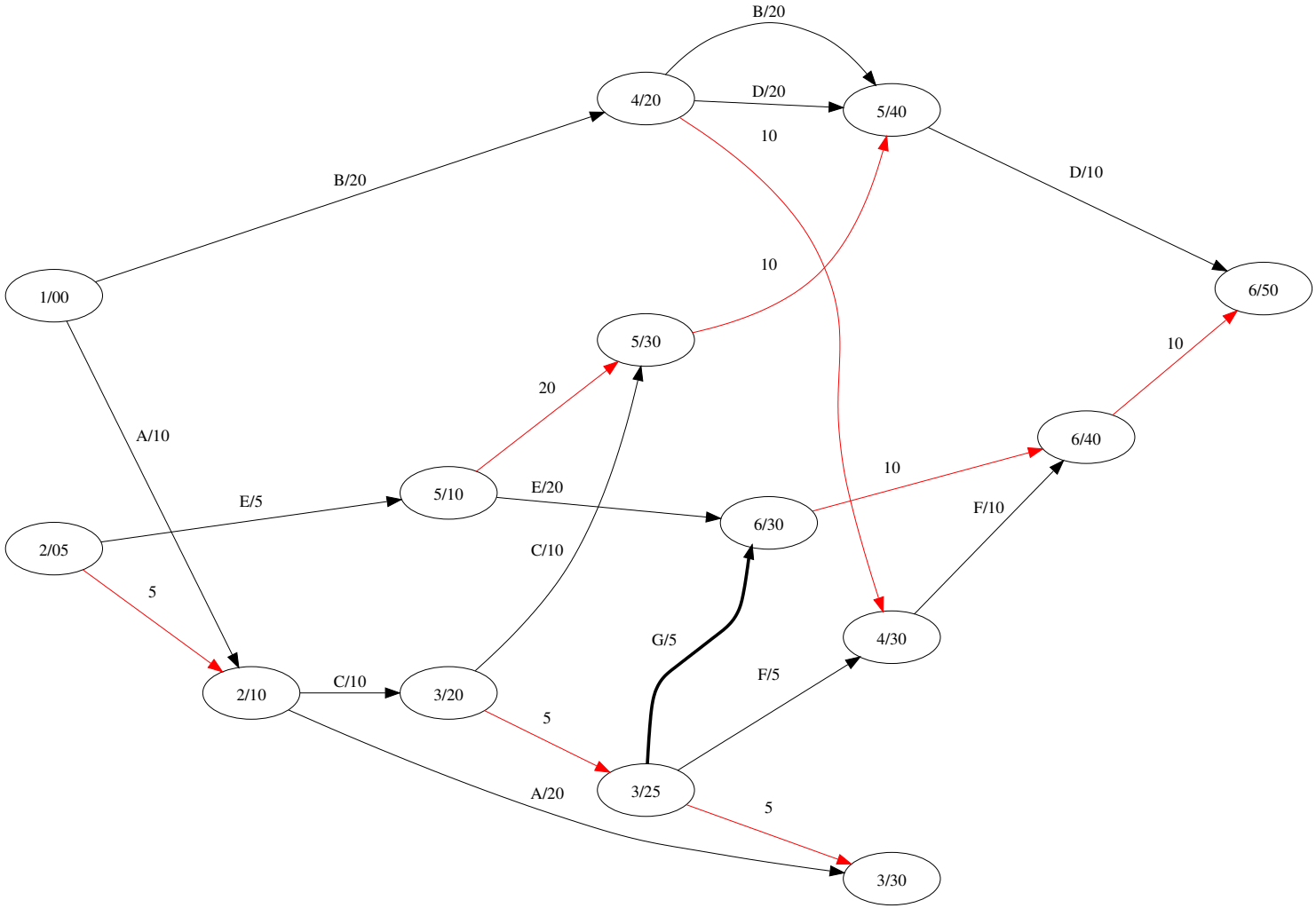
- Previous method only gives us the duration, not the actual path
- *At each iteration, store nodes out of queue with predecessors*

| pred | node |
|---|---|
| - | 1/00 |
| 1/00 | 2/10 |
| 1/00 | 4/20 |
| 2/10 | 3/20 |
| 2/10 | 3/30 |
| 4/20 | 4/30 |
| 4/30 | 6/40 |

- Now retrieve path backwards: 6/40$\rightarrow$4/30$\rightarrow$4/20$\rightarrow$1/00
- and finally invert the path: 1/00$\rightarrow$4/20$\rightarrow$4/30$\rightarrow$6/40

# This ain't the fastest

Suppose there is a bus G with timetable 3/25 $\rightarrow$ 6/30




3/25 is still in the queue ( 5/40 | 3/25 | 5/30 | 6/40 ) at termination, can't reach 6/30

# What did we find?

- In order to find the *fastest* itinerary, must change termination condition

  **instead of stopping when the arrival node is found,**

  let $\tau^*$ be the arrival time of best solution so far

  let $\tau'$ be the minimum time of the nodes in the queue

  **if** $(\tau' \geq \tau^*)$ **then**

  terminate

  **endif**

# What did we find?

- In order to find the *fastest* itinerary, must change termination condition

  **instead of stopping when the arrival node is found,**

  let $\tau^*$ be the arrival time of best solution so far

  let $\tau'$ be the minimum time of the nodes in the queue

  **if** $(\tau' \geq \tau^*)$ **then**

  terminate

  **endif**

- What are the properties of our itinerary?

# What did we find?

- In order to find the *fastest* itinerary, must change termination condition

  **instead of stopping when the arrival node is found,**

  let $\tau^*$ be the arrival time of best solution so far

  let $\tau'$ be the minimum time of the nodes in the queue

  **if** $(\tau' \geq \tau^*)$ **then**

  terminate

  **endif**

- What are the properties of our itinerary?
- We found the **itinerary with fewest changes**

  where “bus, waiting, bus” counts as two changes, not one

# What did we find?

- In order to find the *fastest* itinerary, must change termination condition

  **instead of stopping when the arrival node is found,**

  let $\tau^*$ be the arrival time of best solution so far
  let $\tau'$ be the minimum time of the nodes in the queue
  **if** $(\tau' \geq \tau^*)$ **then**
  terminate
  **endif**

- What are the properties of our itinerary?
- We found the **itinerary with fewest changes**

  where “bus, waiting, bus” counts as two changes, not one

- In order to prove these two statements, we need to formalize our method into an algorithm

# What did we find?

- In order to find the *fastest* itinerary, must change termination condition

  **instead of stopping when the arrival node is found,**

  let $\tau^*$ be the arrival time of best solution so far
  let $\tau'$ be the minimum time of the nodes in the queue
  **if** $(\tau' \geq \tau^*)$ **then**
  terminate
  **endif**

- What are the properties of our itinerary?
- We found the **itinerary with fewest changes**

  where “bus, waiting, bus” counts as two changes, not one

- In order to prove these two statements, we need to formalize our method into an algorithm
- So, we need to describe our “queue” as a data structure


# Queues

# Queue operations

- In our example, we needed our “queue” to be able to perform the following operations:
  - **insert** *an element at the* **end** *of the queue*
  - **retrieve and delete** *the element at the* **beginning** *of the queue*
  - *test if the queue* **is empty**
  - *find the* **size** *of the queue*

# Queue operations


- In our example, we needed our “queue” to be able to perform the following operations:
  - **insert** *an element at the* **end** *of the queue*
  - **retrieve and delete** *the element at the* **beginning** *of the queue*
  - *test if the queue* **is empty**
  - *find the* **size** *of the queue*
- Since these operations were repeated often, we need them to be $O(1)$

# Queue operations

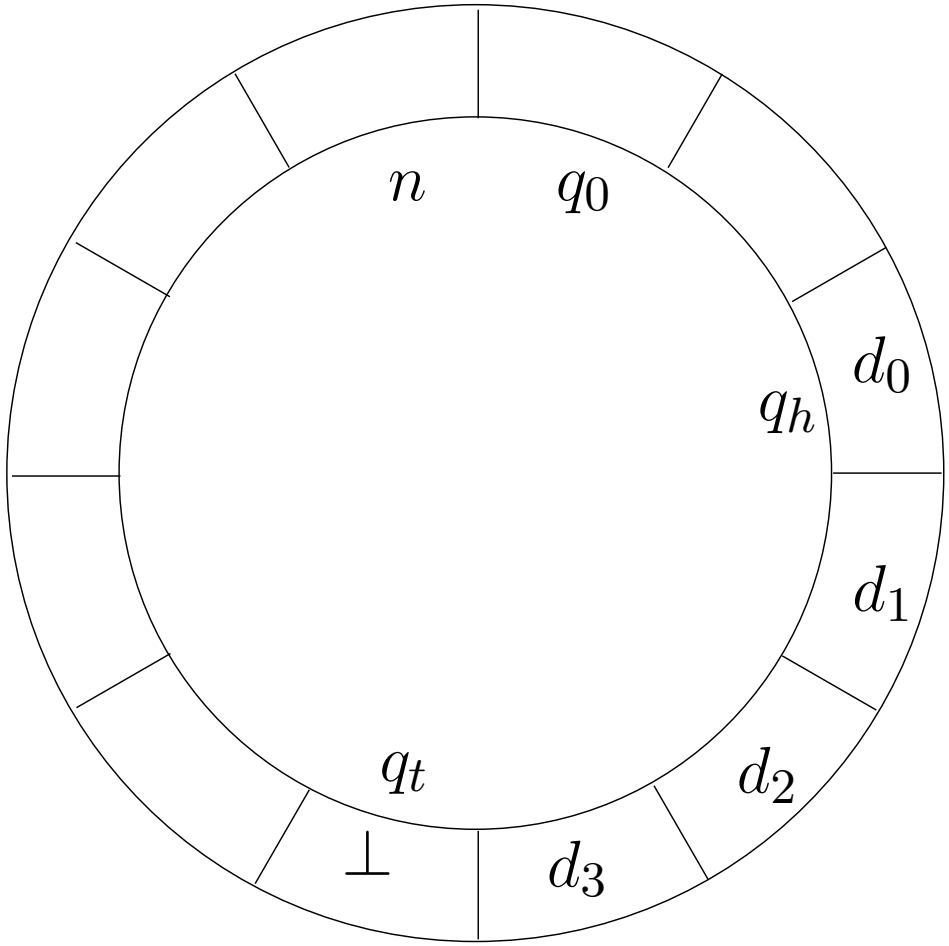
- In our example, we needed our “queue” to be able to perform the following operations:
  - **insert** *an element at the* **end** *of the queue*
  - **retrieve and delete** *the element at the* **beginning** *of the queue*
  - *test if the queue* **is empty**
  - *find the* **size** *of the queue*
- Since these operations were repeated often, we need them to be $O(1)$
- Could implement these using:
  - arrays

    Need to simulate insert/delete using pointers
  - lists

    Each insert/delete involves memory allocation

# Circular arrays

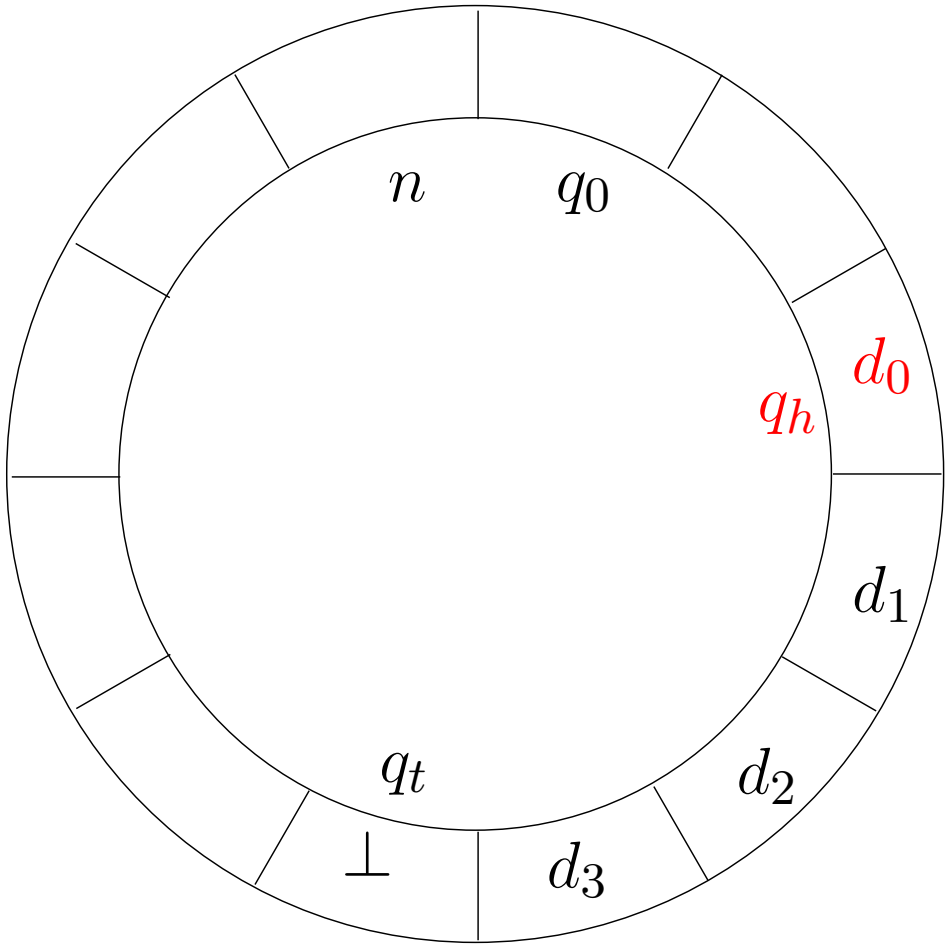
- Implementation using a *circular array* $q$ [Mehlhorn & Sanders' book]
- Uses modular arithmetic (usually pretty fast)




- $q_h = d_0$ (first element of queue)
- $q_t = \perp$ delimits the end of the queue
- $q_i$ is unused for $0 \leq i < h$ and $t < i \leq n$
- actual implementation of a circular array is simply an array; but the behaviour will be different

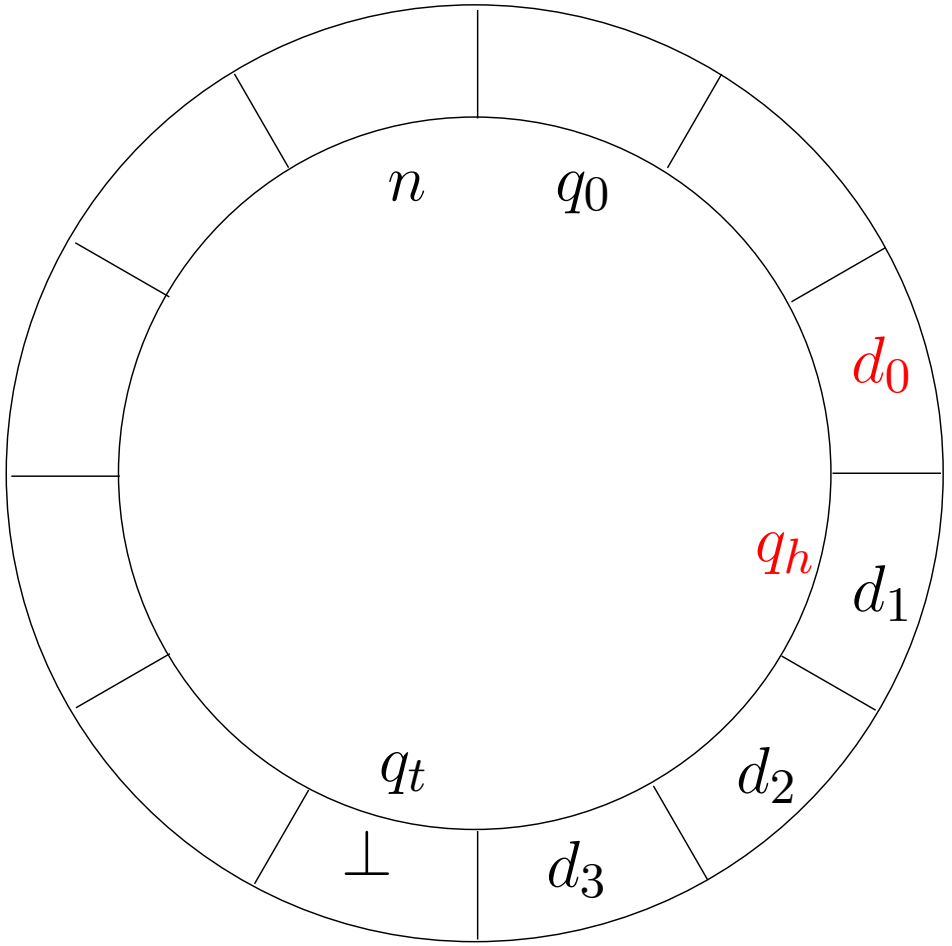
# Read beginning of queue

```
first() {
  return q_h;
}
```



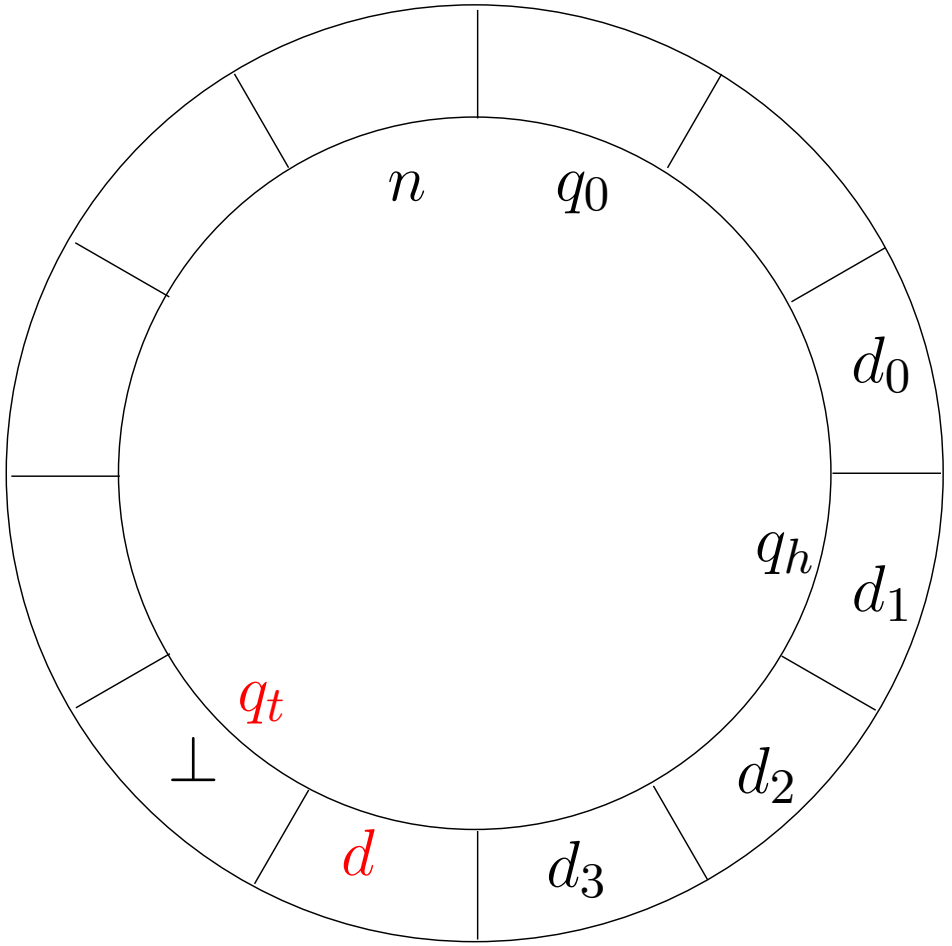
# Delete beginning of queue

```
popFront() {
    p = q_h;
    h = (h + 1) mod (n + 1);
    return p;
}
```

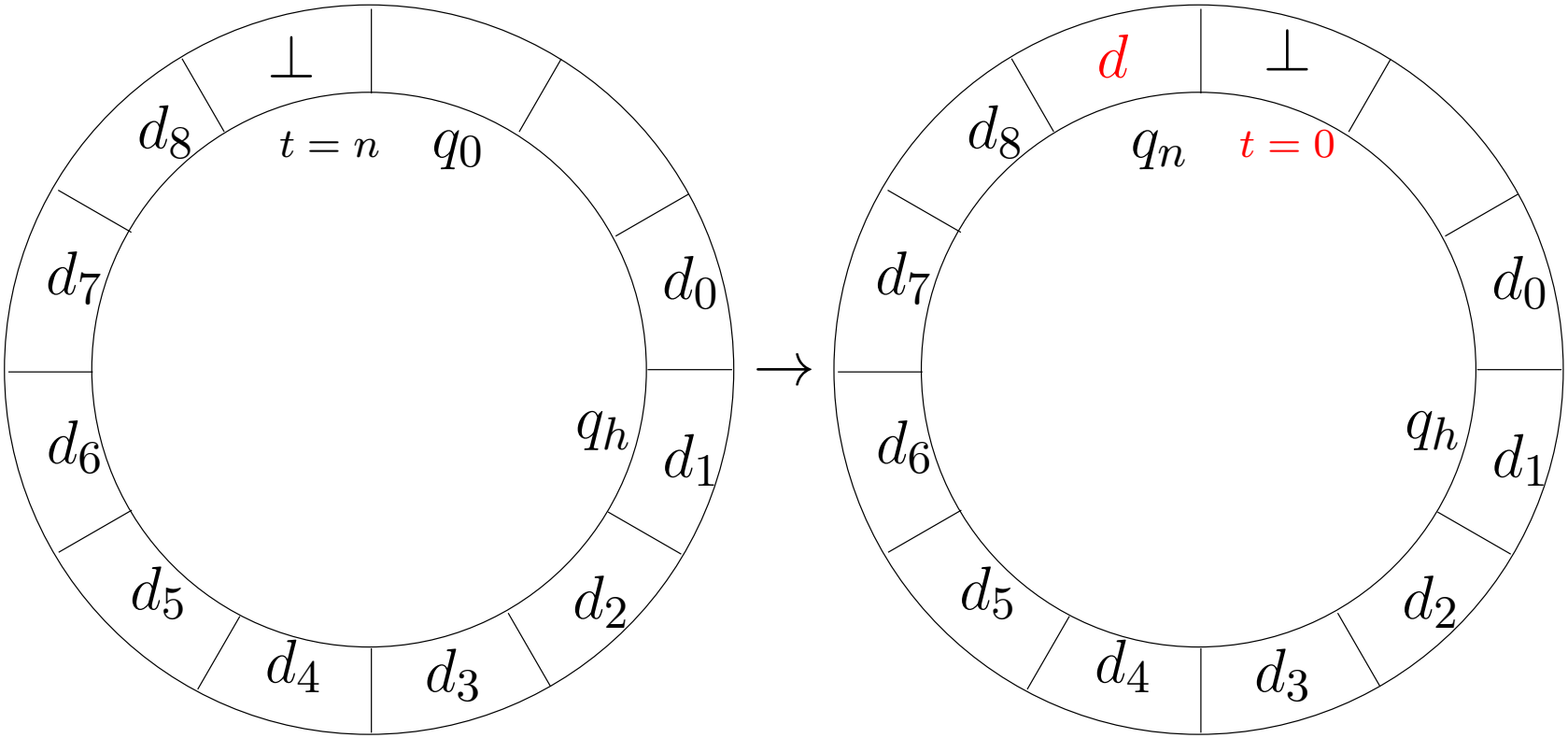


# Insert at the end of queue

```
pushBack(d) {
    assert(size() < n)
    q_t = d;
    t = (t + 1) mod (n + 1);
    q_t = ⊥;
}
```



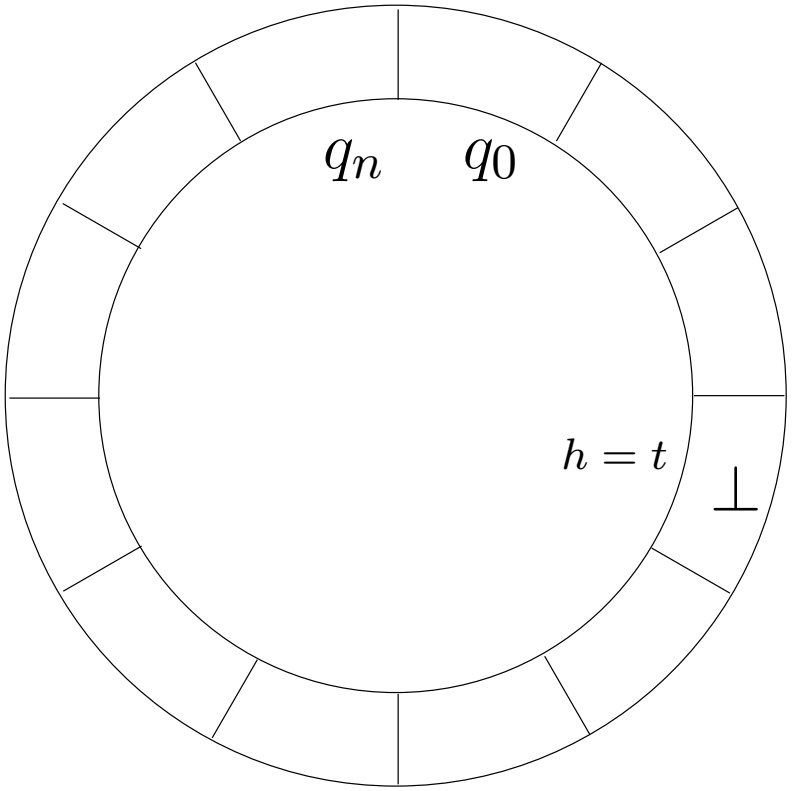
# Insert at the end (case $t = n$)




$t = (t + 1) \bmod (n + 1)$ with $t = n$ implies $t = 0$

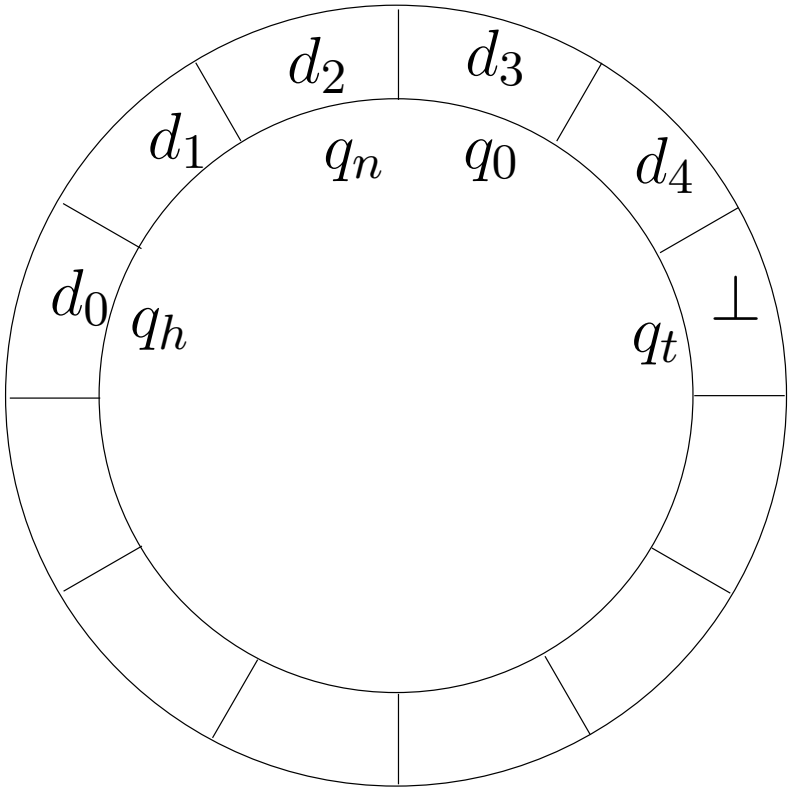
# Size and emptiness

- `isEmpty()`: **if** $(t = h)$ **then return true; else return false;**



# Size and emptiness

- `isEmpty()`: **if** $(t = h)$ **then return true; else return false;**
- `size()`: **return** $(t - h + n + 1) \bmod (n + 1)$;




# BFS

# The BFS algorithm

**Input**: set $V$, binary relation $\sim$ on $V$, and $s \neq t \in V$

1: $(Q, <) = \{s\}$; $L = \{s\}$;
2: **while** $Q \neq \varnothing$ **do**
3: $u = \min_< Q$;
4: $Q \leftarrow Q \setminus \{u\}$;
5: **for** $v \in V$ $(v \sim u \wedge v \notin L)$ **do**
6: **if** $v = t$ **then**
7: **return** "$t$ reachable";
8: **end if**
9: $Q \leftarrow Q \cup \{v\}$, set $v = \max_< Q$;
10: $L \leftarrow L \cup \{v\}$;
11: **end for**
12: **end while**
13: **return** "$t$ unreachable";

# The order on $Q$


- The ordered set $Q$ is implemented as a queue

# The order on $Q$


- The ordered set $Q$ is implemented as a queue
- Every $v \in V$ enters $Q$ as the *maximum* element (i.e. , the last)

# The order on $Q$

- The ordered set $Q$ is implemented as a queue
- Every $v \in V$ enters $Q$ as the *maximum* element (i.e. , the last)
- We only read (and remove) the *minimum* element of $Q$ (i.e. the first)

# The order on $Q$

- The ordered set $Q$ is implemented as a queue
- Every $v \in V$ enters $Q$ as the *maximum* element (i.e. , the last)
- We only read (and remove) the *minimum* element of $Q$ (i.e. the first)
- Every other element of $Q$ is never touched

# The order on $Q$

- The ordered set $Q$ is implemented as a queue
- Every $v \in V$ enters $Q$ as the *maximum* element (i.e. , the last)
- We only read (and remove) the *minimum* element of $Q$ (i.e. the first)
- Every other element of $Q$ is never touched
- The relative order of a consecutive subsequence $u_1, \ldots, u_h$ of $Q$ is unchanged

# The order on $Q$

- The ordered set $Q$ is implemented as a queue
- Every $v \in V$ enters $Q$ as the *maximum* element (i.e. , the last)
- We only read (and remove) the *minimum* element of $Q$ (i.e. the first)
- Every other element of $Q$ is never touched
- The relative order of a consecutive subsequence $u_1, \ldots, u_h$ of $Q$ is unchanged
- Also, by the test $v \notin L$ at Step 5, we have:
  Thm. 1
  No element of $V$ enters $Q$ more than once

# A node hierarchy

- Consider a function $\alpha : V \to \mathbb{N}$ defined as follows:

  at Step 1, let $\alpha(s) = 0$

  at Step 9, let $\alpha(v) = \alpha(u) + 1$

- This ranks the elements of $V$ by distance from $s$ in terms of relation pairs
- E.g. if $s \sim u$, then $u$'s distance from $s$ is 1
  if $s \sim u \sim v$, $v$'s distance from $s$ is 2

# The BFS, again

1: $(Q, <) = \{s\}$; $L = \{s\}$;
2: $\alpha(s) = 0$;
3: **while** $Q \neq \varnothing$ **do**
4: $u = \min_< Q$;
5: $Q \leftarrow Q \setminus \{u\}$;
6: **for** $v \in V$ $(v \sim u \wedge v \notin L)$ **do**
7: $\alpha(v) = \alpha(u) + 1$;
8: **if** $v = t$ **then**
9: **return** "$t$ reachable";
10: **end if**
11: $Q \leftarrow Q \cup \{v\}$, set $v = \max_< Q$;
12: $L \leftarrow L \cup \{v\}$;
13: **end for**
14: **end while**
15: **return** "$t$ unreachable";

# Basic results

We have the following results (try and prove them):

Thm. 2

If $(s, v_1, \ldots, v_k)$ is any itinerary found by BFS, $\alpha(v_k) = k$

# Basic results

We have the following results (try and prove them):

Thm. 2

If $(s, v_1, \ldots, v_k)$ is any itinerary found by BFS, $\alpha(v_k) = k$

Thm. 3

If $\alpha(u) < \alpha(v)$, then $u$ enters $Q$ before $v$ does

# Basic results

We have the following results (try and prove them):

Thm. 2

If $(s, v_1, \ldots, v_k)$ is any itinerary found by BFS, $\alpha(v_k) = k$

Thm. 3

If $\alpha(u) < \alpha(v)$, then $u$ enters $Q$ before $v$ does

Thm. 4

No itinerary found by BFS has repeated elements

# Basic results

We have the following results (try and prove them):

Thm. 2

If $(s, v_1, \ldots, v_k)$ is any itinerary found by BFS, $\alpha(v_k) = k$

Thm. 3

If $\alpha(u) < \alpha(v)$, then $u$ enters $Q$ before $v$ does

Thm. 4

No itinerary found by BFS has repeated elements

Thm. 5

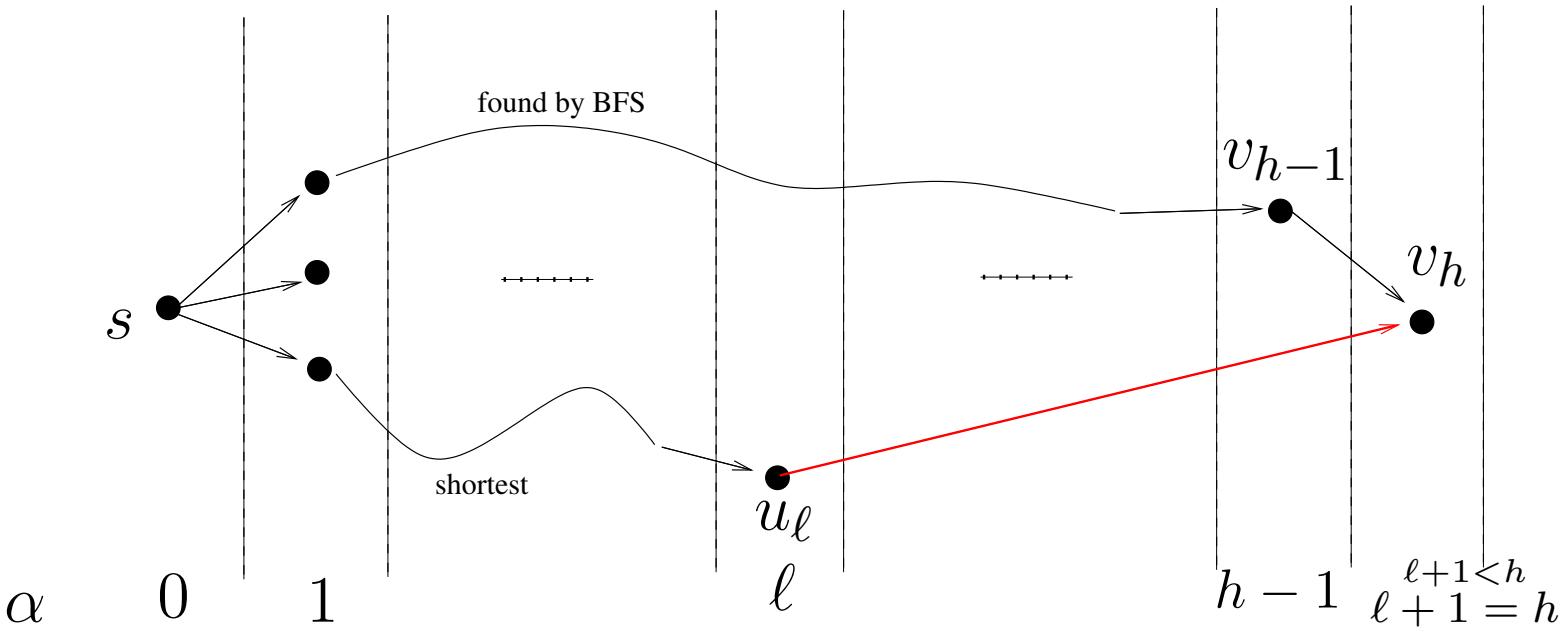
The function $\alpha$ is well defined

# Fewest changes

- Aim to prove that **BFS finds an itinerary with fewest changes**
- **Remark**: the number of changes in an itinerary is the same as the number of nodes in that itinerary, hence: Thm.

BFS finds a shortest itinerary

*Idea of proof:*



# The proof

Thm.

BFS finds a shortest itinerary (in terms of number of nodes)

Proof

Let $R = (s, v_1, \ldots, v_k = t)$ be the itinerary found by BFS: suppose it is not shortest. Let $h \leq k$ be the smallest index such that $R' = (s, \ldots, v_{h-1}, v_h)$ is not shortest from $s \to v_h$. By Thm. 2, $\alpha(v_h) = h$. Since this itinerary is not shortest, there must be a different itinerary $P = (s, u_1, \ldots, u_\ell, v_h)$ which is shortest: then necessarily we have $\ell + 1 < h$, hence $\ell < h$. By Thm. 2 again and induction we have $\alpha(u_\ell) = \ell$; moreover by Thm. 3 $u_\ell$ enters $Q$ before $v_h$, so BFS finds the itinerary $P$ before $R'$. Now $u_\ell \sim v_{h+1}$ yields $\alpha(v_h) = \ell + 1 < h = \alpha(v_h)$, contradiction.

# Finding all shortest itineraries

- Delete Steps 8-10
- All elements in $V$ enter and exit $Q$
- Finds shortest itineraries from $s$ to all elements of $V$

**WARNING**: BFS will *not* find shortest paths in a weighted graph unless all the arc costs are 1

# What about fastest?

- Every time we insert $v$ at the end of $Q$, we also record the arrival time at $v$ using the travelling information about the relation $u \sim v$

# What about fastest?

- Every time we insert $v$ at the end of $Q$, we also record the arrival time at $v$ using the travelling information about the relation $u \sim v$
- We keep track of the *minimum time* $\tau'$ over all elements of $Q$

# What about fastest?

- Every time we insert $v$ at the end of $Q$, we also record the arrival time at $v$ using the travelling information about the relation $u \sim v$
- We keep track of the *minimum time* $\tau'$ over all elements of $Q$
- Whenever we extract the arrival node $t$ from $Q$, we have a possible itinerary $s \to t$; among these itineraries, we keep track best arrival time $\tau^*$ to $t$ so far

# What about fastest?

- Every time we insert $v$ at the end of $Q$, we also record the arrival time at $v$ using the travelling information about the relation $u \sim v$
- We keep track of the *minimum time* $\tau'$ over all elements of $Q$
- Whenever we extract the arrival node $t$ from $Q$, we have a possible itinerary $s \to t$; among these itineraries, we keep track best arrival time $\tau^*$ to $t$ so far
- We update $\tau^*$ whenever we find a better itinerary $s \to t$

# What about fastest?

- Every time we insert $v$ at the end of $Q$, we also record the arrival time at $v$ using the travelling information about the relation $u \sim v$
- We keep track of the *minimum time* $\tau'$ over all elements of $Q$
- Whenever we extract the arrival node $t$ from $Q$, we have a possible itinerary $s \to t$; among these itineraries, we keep track best arrival time $\tau^*$ to $t$ so far
- We update $\tau^*$ whenever we find a better itinerary $s \to t$
- As soon as $\tau' \geq \tau^*$, we know we have a shortest itinerary (why?)


# Implementation

# A possible implementation

```
1: L is initialized to {s};
2: Q is an empty queue;
3: α(s) = 0;
4: Q.pushBack(s);
5: while ¬(Q.isEmpty()) do
6:    u = Q.popFront();
7:    for v ∈ V (v ∼ u ∧ v ∉ L) do
8:       α(v) = α(u) + 1
9:       if v = t then
10:         return α(v);
11:      end if
12:      Q.pushBack(v);
13:      L.pushBack(v);
14:   end for
15: end while
16: return "t unreachable";
```

```
1: L = {s};
2: (Q, <) = ∅;
3: α(s) = 0;
4: (Q, <) = {s};
5: while Q ≠ ∅ do
6:    u = min_< Q; Q ← Q ∖ {u};
7:    for v ∈ V (v ∼ u ∧ v ∉ L) do
8:       α(v) = α(u) + 1
9:       if v = t then
10:         return α(v);
11:      end if
12:      Q ← Q ∪ {v} (v = max_< Q);
13:      L ← L ∪ {v};
14:   end for
15: end while
16: return "t unreachable";
```

# Its problems

- How efficiently can you implement Step 7?

**for** $v \in V\ (v \sim u \wedge v \notin L)$ **do**

# Its problems


- How efficiently can you implement Step 7?

$$\boxed{\textbf{for } v \in V \ (v \sim u \wedge v \notin L) \textbf{ do}}$$

- Looping over $V$: worst case $O(|V|)$

# Its problems

- How efficiently can you implement Step 7?

$$\boxed{\textbf{for } v \in V \ (v \sim u \wedge v \notin L) \textbf{ do}}$$

- Looping over $V$: worst case $O(|V|)$
- For each $v$, check whether $v \sim u$
  - with a jagged array, the $u$-th row can have size at worst $O(|V|)$, if every $v$ is in relation with $u$
  - if we keep a map $A : V \times V \to \{0, 1\}$ such that $A(u, v) = 1$ whenever $u \sim v$ and 0 otherwise, we reduce this to $O(1)$

# Its problems

- How efficiently can you implement Step 7?

$$\boxed{\textbf{for } v \in V \ (v \sim u \wedge v \notin L) \textbf{ do}}$$

- Looping over $V$: worst case $O(|V|)$
- For each $v$, check whether $v \sim u$
  - with a jagged array, the $u$-th row can have size at worst $O(|V|)$, if every $v$ is in relation with $u$
  - if we keep a map $A : V \times V \to \{0, 1\}$ such that $A(u, v) = 1$ whenever $u \sim v$ and 0 otherwise, we reduce this to $O(1)$
- For each $v$, also check whether $v \notin L$: worst case $O(|V|)$ (when most nodes of $V$ have been put into $L$)

# Its problems

- How efficiently can you implement Step 7?

$$\boxed{\textbf{for } v \in V \ (v \sim u \wedge v \notin L) \textbf{ do}}$$

- Looping over $V$: worst case $O(|V|)$
- For each $v$, check whether $v \sim u$
  - with a jagged array, the $u$-th row can have size at worst $O(|V|)$, if every $v$ is in relation with $u$
  - if we keep a map $A : V \times V \to \{0, 1\}$ such that $A(u, v) = 1$ whenever $u \sim v$ and 0 otherwise, we reduce this to $O(1)$
- For each $v$, also check whether $v \notin L$: worst case $O(|V|)$ (when most nodes of $V$ have been put into $L$)
- A worst case complexity of $O(|V|(1 + |V|)) = O(|V|^2)$

# Its problems

- How efficiently can you implement Step 7?

$$\boxed{\textbf{for } v \in V \ (v \sim u \wedge v \notin L) \textbf{ do}}$$

- Looping over $V$: worst case $O(|V|)$
- For each $v$, check whether $v \sim u$
  - with a jagged array, the $u$-th row can have size at worst $O(|V|)$, if every $v$ is in relation with $u$
  - if we keep a map $A : V \times V \to \{0, 1\}$ such that $A(u, v) = 1$ whenever $u \sim v$ and 0 otherwise, we reduce this to $O(1)$
- For each $v$, also check whether $v \notin L$: worst case $O(|V|)$ (when most nodes of $V$ have been put into $L$)
- A worst case complexity of $O(|V|(1 + |V|)) = O(|V|^2)$

Repeated in the external loop: get $O(|V|^3)$ overall: **ugh!**

# A more efficient alternative

- We initialize the node ranking function $\alpha$ so that $\alpha(u) = |V| + 1$ for all $u \in V \setminus \{s\}$ before Step 5

# A more efficient alternative

- We initialize the node ranking function $\alpha$ so that $\alpha(u) = |V| + 1$ for all $u \in V \setminus \{s\}$ before Step 5
- Remark that, by induction from $\alpha(s) = 0$, whenever $\alpha(v)$ is updated at Step 8 its value is always $\leq |V|$

# A more efficient alternative

- We initialize the node ranking function $\alpha$ so that $\alpha(u) = |V| + 1$ for all $u \in V \setminus \{s\}$ before Step 5
- Remark that, by induction from $\alpha(s) = 0$, whenever $\alpha(v)$ is updated at Step 8 its value is always $\leq |V|$
- Since $v$ is inserted into $L$ after $\alpha(v)$ is updated at Step 8, for each $v \in V$ we have that $v \in L$ if and only if $\alpha(v) \leq |V|$

# A more efficient alternative

- We initialize the node ranking function $\alpha$ so that $\alpha(u) = |V| + 1$ for all $u \in V \setminus \{s\}$ before Step 5
- Remark that, by induction from $\alpha(s) = 0$, whenever $\alpha(v)$ is updated at Step 8 its value is always $\leq |V|$
- Since $v$ is inserted into $L$ after $\alpha(v)$ is updated at Step 8, for each $v \in V$ we have that $v \in L$ if and only if $\alpha(v) \leq |V|$
- Change loop at Step 7 as follows:

**for** $v \in V$ $(v \sim u \wedge \alpha(v) = |V| + 1)$ **do**

# A more efficient alternative

- We initialize the node ranking function $\alpha$ so that $\alpha(u) = |V| + 1$ for all $u \in V \setminus \{s\}$ before Step 5
- Remark that, by induction from $\alpha(s) = 0$, whenever $\alpha(v)$ is updated at Step 8 its value is always $\leq |V|$
- Since $v$ is inserted into $L$ after $\alpha(v)$ is updated at Step 8, for each $v \in V$ we have that $v \in L$ if and only if $\alpha(v) \leq |V|$
- Change loop at Step 7 as follows:

$$\textbf{for } v \in V \ (v \sim u \wedge \alpha(v) = |V| + 1) \textbf{ do}$$

- Now worst-case complexity is $O(|V|)$ (table look-up in $O(1)$)

# A more efficient alternative

- We initialize the node ranking function $\alpha$ so that $\alpha(u) = |V| + 1$ for all $u \in V \setminus \{s\}$ before Step 5
- Remark that, by induction from $\alpha(s) = 0$, whenever $\alpha(v)$ is updated at Step 8 its value is always $\leq |V|$
- Since $v$ is inserted into $L$ after $\alpha(v)$ is updated at Step 8, for each $v \in V$ we have that $v \in L$ if and only if $\alpha(v) \leq |V|$
- Change loop at Step 7 as follows:

  **for** $v \in V$ $(v \sim u \wedge \alpha(v) = |V| + 1)$ **do**
- Now worst-case complexity is $O(|V|)$ (table look-up in $O(1)$)

Hence, we have $O(|V|^2)$ overall

# A better worst-case analysis

- The internal loop

$$\textbf{for } v \in V \ (v \sim u \wedge \alpha(v) = |V| + 1) \textbf{ do}$$

only loops over relation pairs $(u, v)$

# A better worst-case analysis

- The internal loop

$$\textbf{for } v \in V \ (v \sim u \wedge \alpha(v) = |V| + 1) \textbf{ do}$$

  only loops over relation pairs $(u, v)$

- Because $u$ is never considered more than once by Thm. 1, it follows that no pair $(u, v)$ is ever considered more than once over all iterations w.r.t. both loops

# A better worst-case analysis

- The internal loop

$$\textbf{for } v \in V \ (v \sim u \wedge \alpha(v) = |V| + 1) \textbf{ do}$$

  only loops over relation pairs $(u, v)$

- Because $u$ is never considered more than once by Thm. 1, it follows that no pair $(u, v)$ is ever considered more than once over all iterations w.r.t. both loops
- At worst, the instruction $Q.\texttt{pushBack}(v)$ can only be repeated as many times as there are pairs in the relation $\sim$

# A better worst-case analysis

- The internal loop

$$\textbf{for } v \in V \ (v \sim u \wedge \alpha(v) = |V| + 1) \textbf{ do}$$

  only loops over relation pairs $(u, v)$

- Because $u$ is never considered more than once by Thm. 1, it follows that no pair $(u, v)$ is ever considered more than once over all iterations w.r.t. both loops
- At worst, the instruction $Q.\texttt{pushBack}(v)$ can only be repeated as many times as there are pairs in the relation $\sim$
- Moreover, we execute the body of the outer loop at least $|V|$ times

# A better worst-case analysis

- The internal loop

$$\textbf{for } v \in V \ (v \sim u \wedge \alpha(v) = |V| + 1) \textbf{ do}$$

  only loops over relation pairs $(u, v)$
- Because $u$ is never considered more than once by Thm. 1, it follows that no pair $(u, v)$ is ever considered more than once over all iterations w.r.t. both loops
- At worst, the instruction $Q.\texttt{pushBack}(v)$ can only be repeated as many times as there are pairs in the relation $\sim$
- Moreover, we execute the body of the outer loop at least $|V|$ times
- Asymptotically, we cannot compare $n, m$: it depends on the graph

# A better worst-case analysis

- The internal loop

$$\textbf{for } v \in V \ (v \sim u \wedge \alpha(v) = |V| + 1) \textbf{ do}$$

  only loops over relation pairs $(u, v)$
- Because $u$ is never considered more than once by Thm. 1, it follows that no pair $(u, v)$ is ever considered more than once over all iterations w.r.t. both loops
- At worst, the instruction $Q.\texttt{pushBack}(v)$ can only be repeated as many times as there are pairs in the relation $\sim$
- Moreover, we execute the body of the outer loop at least $|V|$ times
- Asymptotically, we cannot compare $n, m$: it depends on the graph
- $\Rightarrow$ The worst-case complexity of BFS is then $O(|V| + | \sim |)$


# BFS: a French publication record

# History of BFS


- BFS was “discovered” by several people, and no-one quite knows who was the first
- The first *official publication* BFS is usually pointed out to be a paper written by E.F. Moore, called *The shortest path through a maze*, which appeared in the proceedings of a 1959 conference
- In fact, the book *Théorie des graphes et ses applications* by Claude Berge, published in 1958, contains a description of BFS applied to finding shortest paths

# Berge's problem statement

CHAPITRE 7

## LE PROBLÈME DU PLUS COURT CHEMIN

### LES PROCESSUS A ÉTAPES

Étant donné un graphe $(X, \Gamma)$ et deux sommets $a$ et $b$, on se pose les problèmes suivants :

**Problème 1.** — *Trouver un chemin du graphe allant de $a$ à $b$.*

**Problème 2.** — *Trouver un chemin de longueur minimum allant de $a$ à $b$.*
Le problème 2 contient le problème 1.

# Berge's algorithm

**Algorithme pour le problème 2.** — Par une procédure itérative, on donnera de proche en proche à chaque sommet $x$ une cote égale à la longueur du plus court chemin allant de $a$ à $x$ :

1° On marque le sommet $a$ avec l'indice 0.

2° Si tous les sommets marqués avec l'indice $m$ forment un ensemble $A(m)$ connu, on marque avec l'indice $m+1$ les sommets de l'ensemble :

$$A(m+1) = \{ x \,/\, x \in \Gamma A(m),\ x \notin A(k) \quad \text{pour tout} \quad k \leqslant m \}$$

3° On s'arrête dès que le sommet $b$ a été marqué ; si $b \in A(m)$, on considérera des sommets $b_1, b_2, \ldots$, tels que :

$$\begin{aligned} b_1 &\in A(m-1), & b_1 &\in \Gamma^{-1} b \\ b_2 &\in A(m-2), & b_2 &\in \Gamma^{-1} b_1 \\ &\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ b_m &\in A(0), & b_m &\in \Gamma^{-1} b_{m-1}. \end{aligned}$$

Le chemin $\mu = [a = b_m, b_{m-1}, \ldots, b_1, b]$ est le chemin cherché.


# End of Lecture 2