INF421, Lecture 2
Queues, BFS
Hashing

Leo Liberti

LIX, École Polytechnique, France

Course

- **Objective**: teach notions AND develop intelligence
- **Evaluation**: TP noté en salle info, Contrôle à la fin. Note: $\max(CC, \frac{3}{4}CC + \frac{1}{4}TP)$
- **Organization**: fri 31/8, 7/9, 14/9, 21/9, 28/9, 5/10, 12/10, 19/10, 26/10, amphi 1030-12 (Arago), TD 1330-1530, 1545-1745 (SI:30-34)
- **Books**:
  1. K. Mehlhorn & P. Sanders, *Algorithms and Data Structures*, Springer, 2008
  2. D. Knuth, *The Art of Computer Programming*, Addison-Wesley, 1997
  3. G. Dowek, *Les principes des langages de programmation*, Editions de l'X, 2008
  4. Ph. Baptiste & L. Maranget, *Programmation et Algorithmique*, Ecole Polytechnique (Polycopié), 2006
- **Website**: `www.enseignement.polytechnique.fr/informatique/INF421`
- **Blog**: `inf421.wordpress.com`
- **Contact**: `liberti@lix.polytechnique.fr` (e-mail subject: INF421)

Lecture summary

- Queues and BFS: motivating example
- Queues
- BFS
- Hashing

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

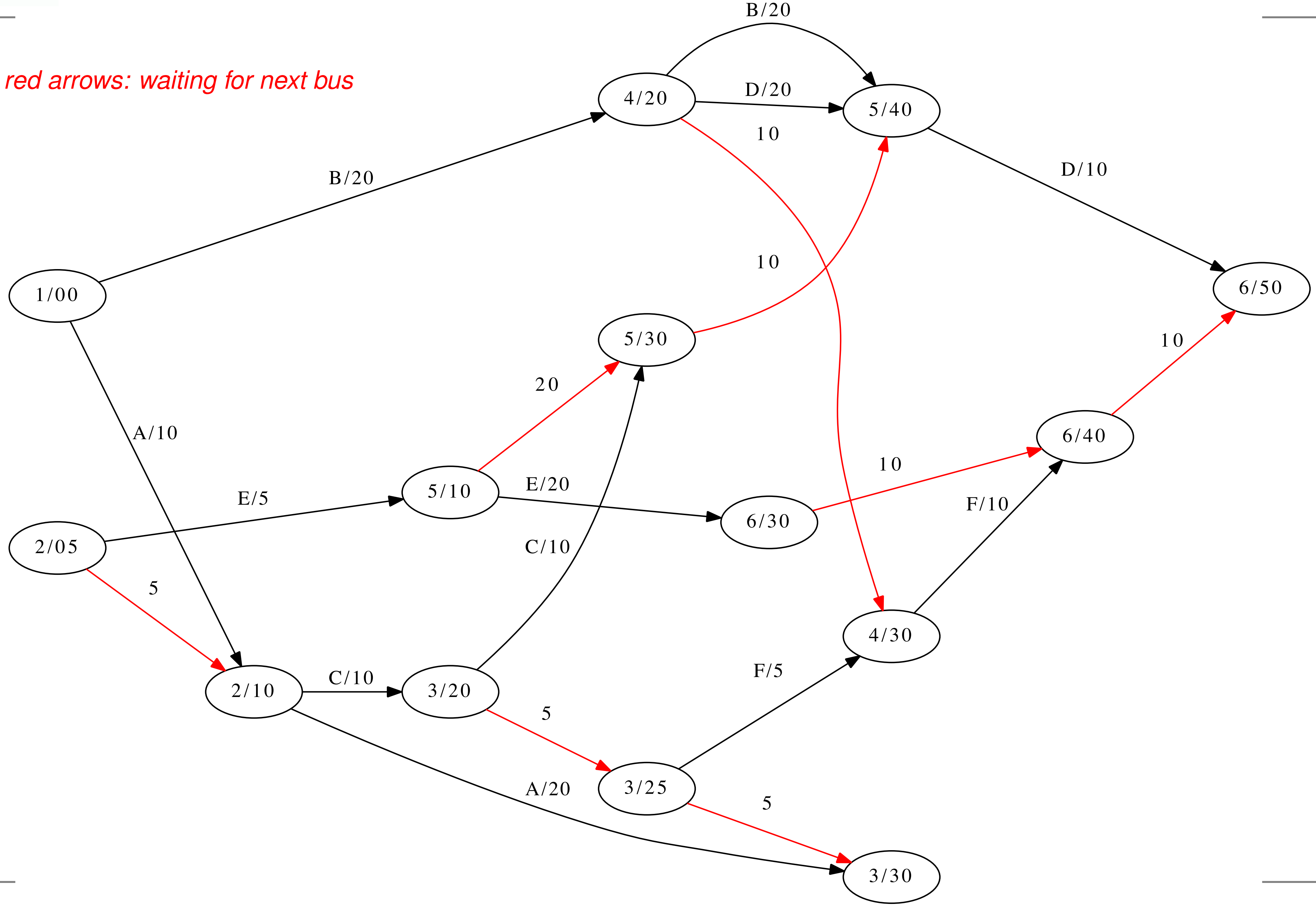
E	
2	h:05
5	h:10
6	h:30

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

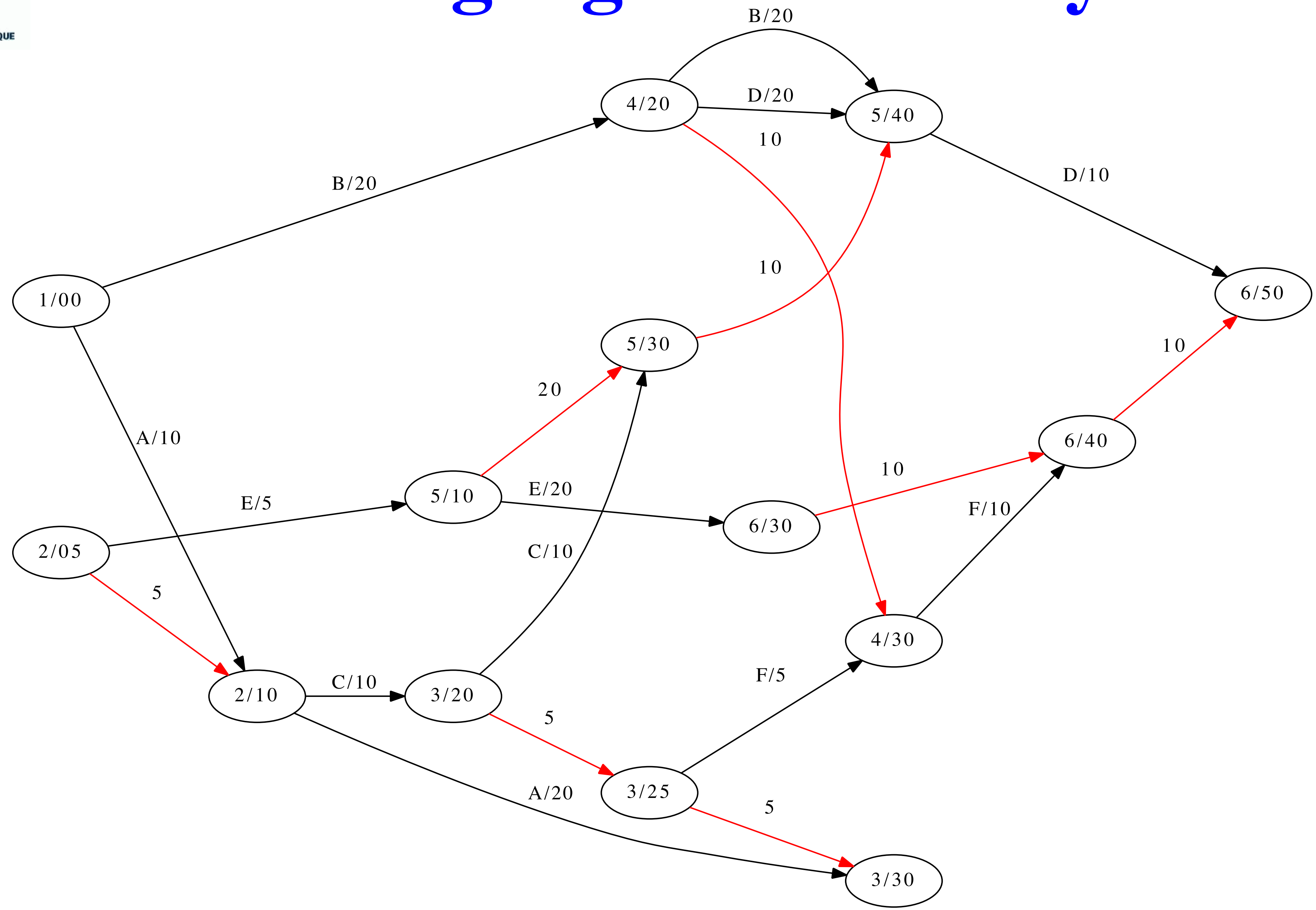
Find a convenient itinerary from 1 to 6, leaving at h:00?

The event graph

red arrows: waiting for next bus

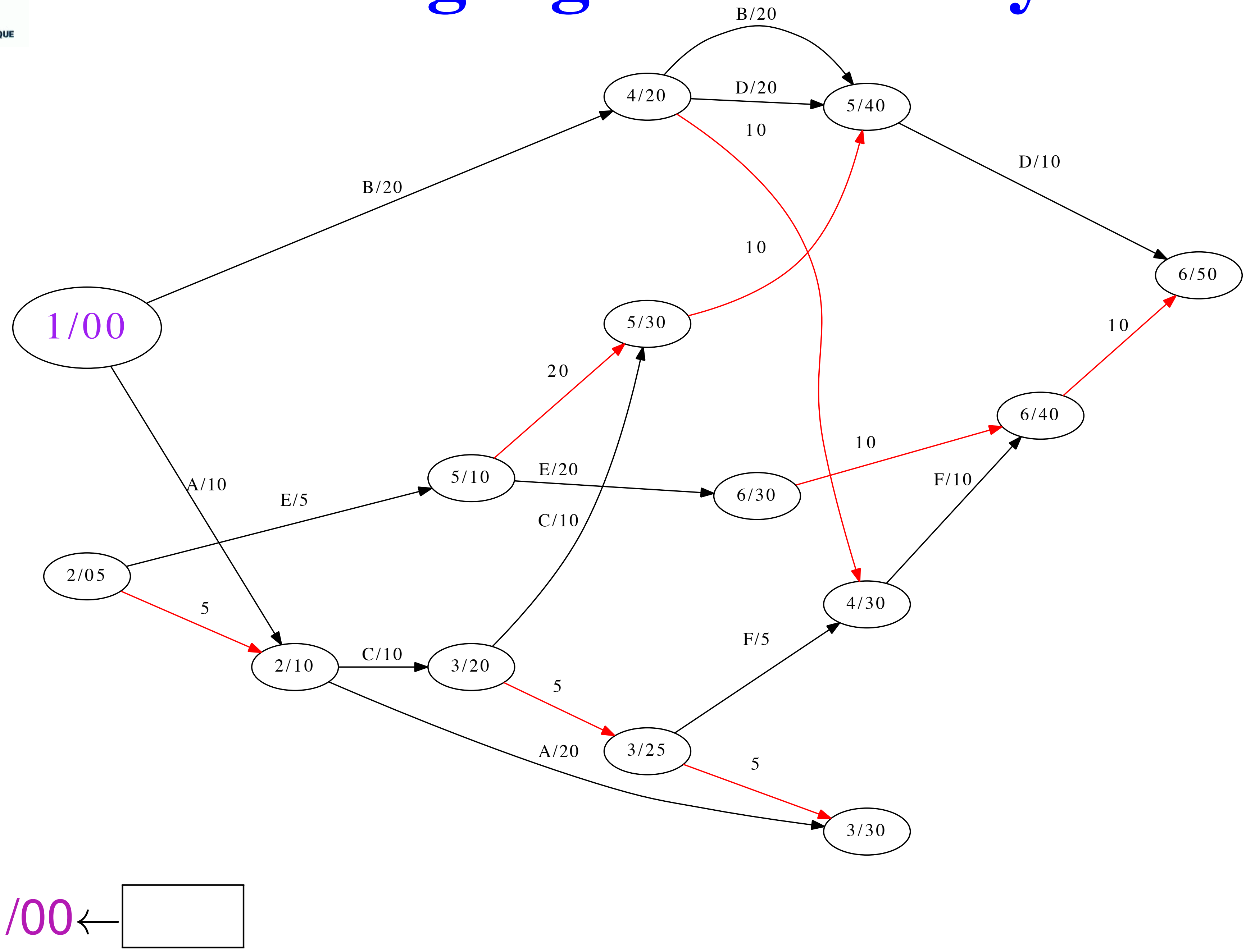


Finding a good itinerary

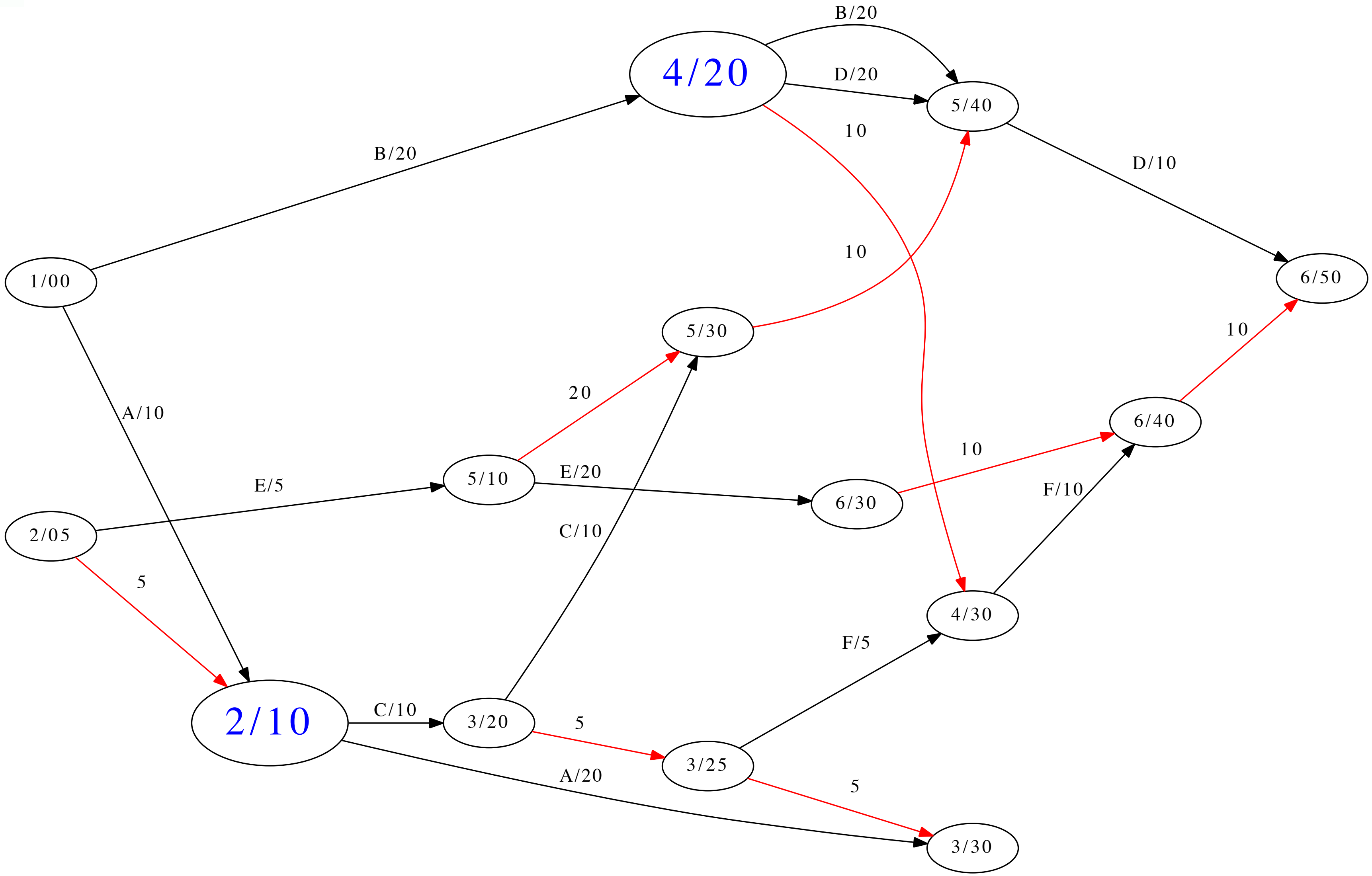


1/00

Finding a good itinerary

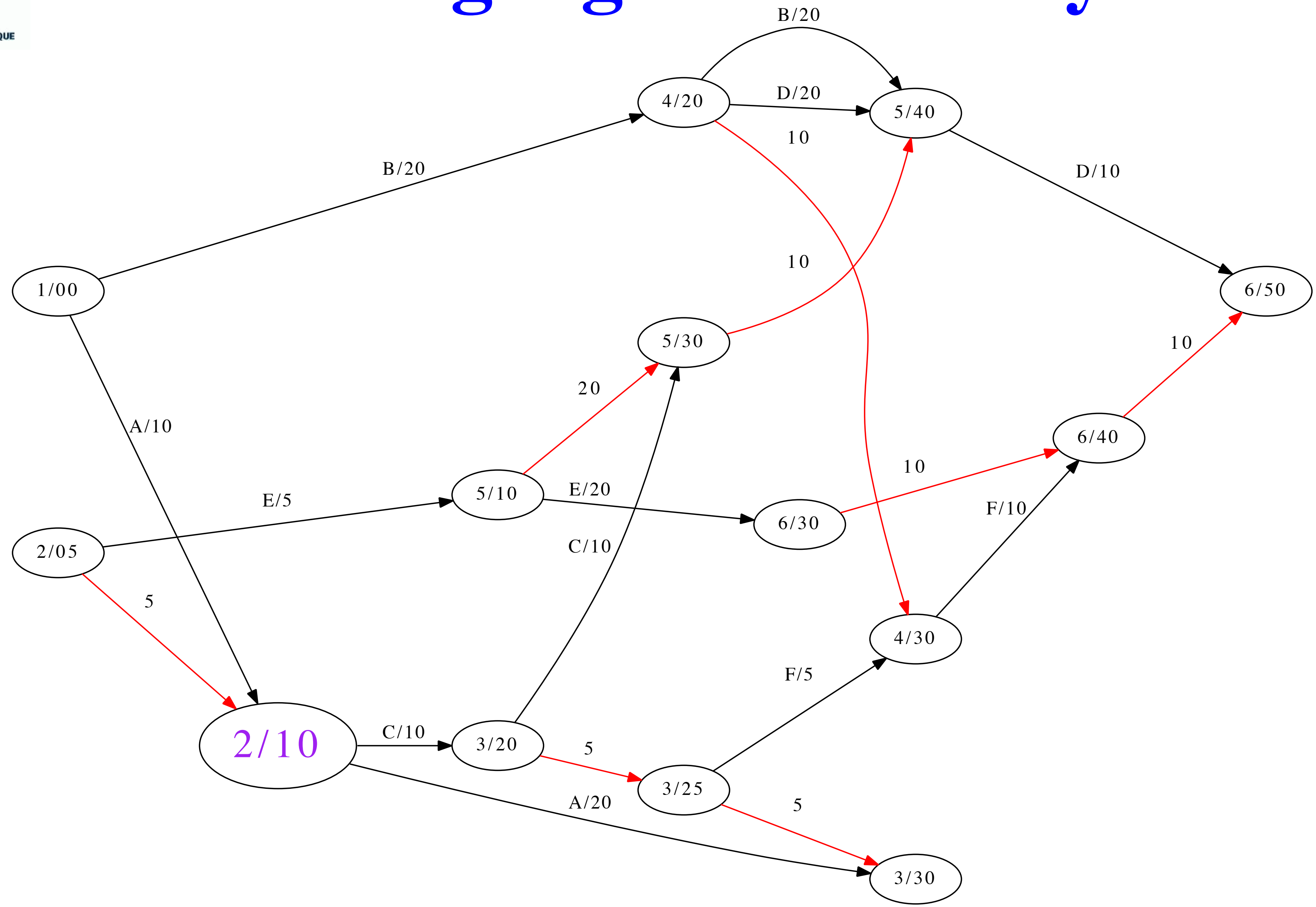


Finding a good itinerary



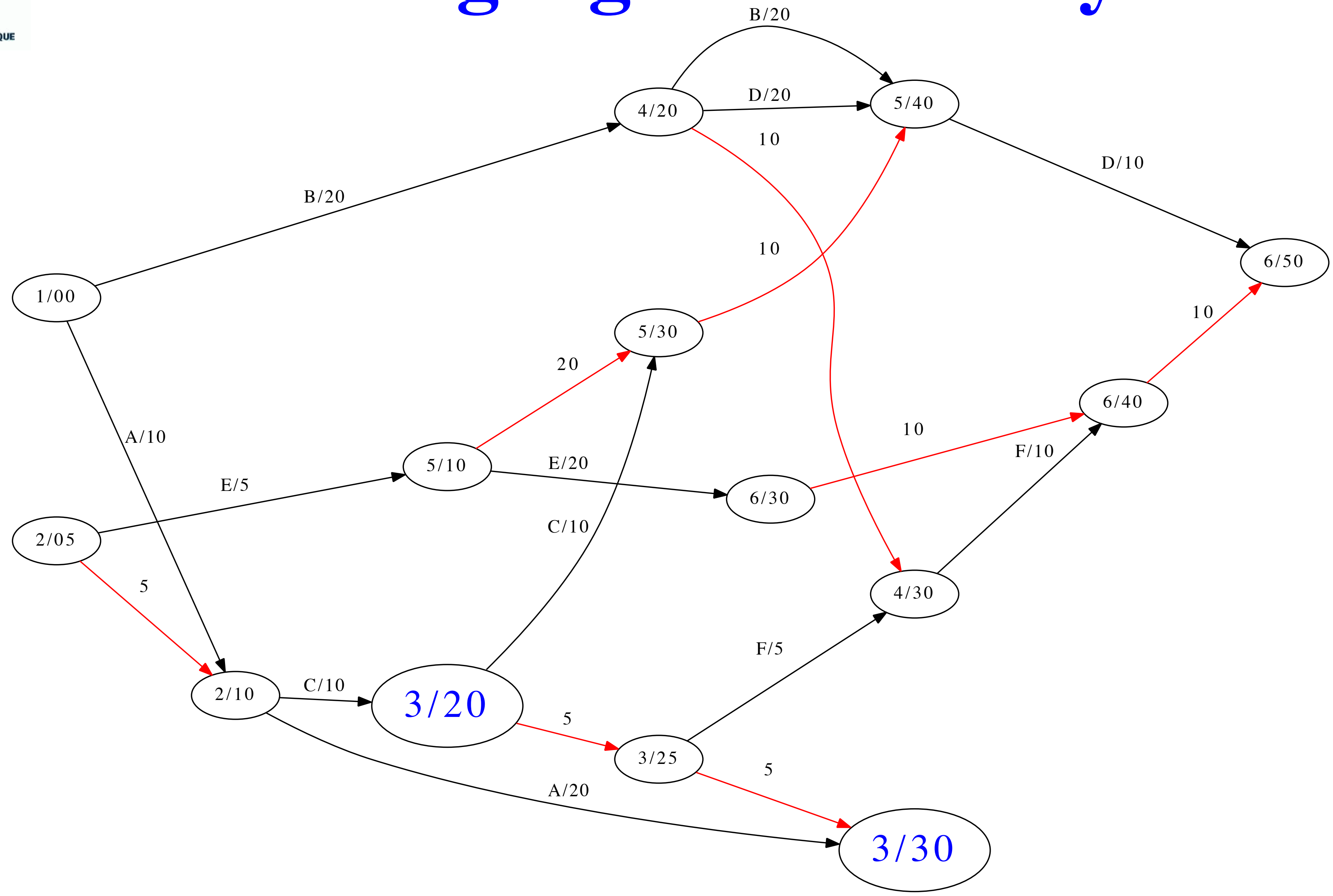
2/10	4/20

Finding a good itinerary



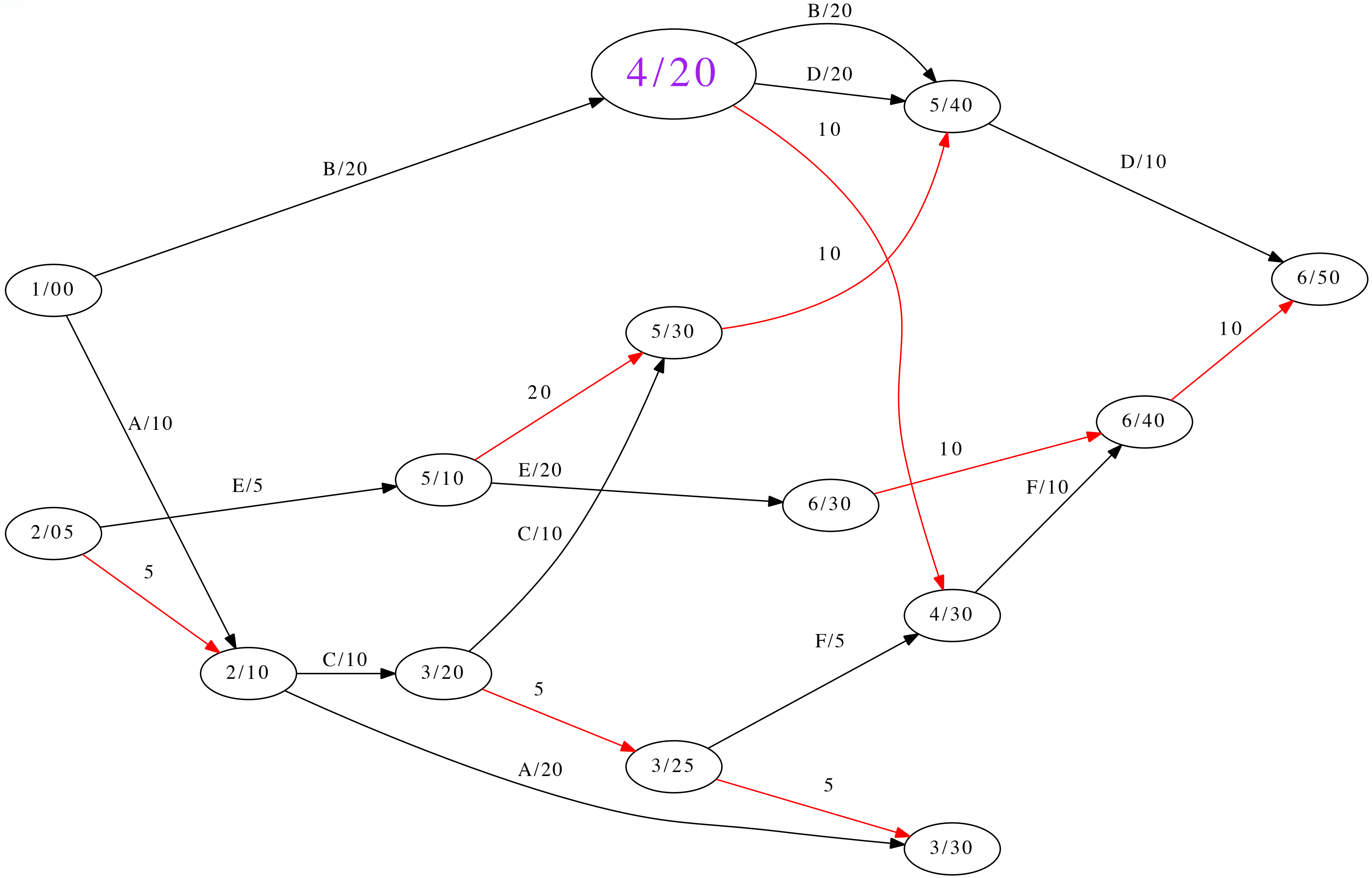
2/10←4/20

Finding a good itinerary



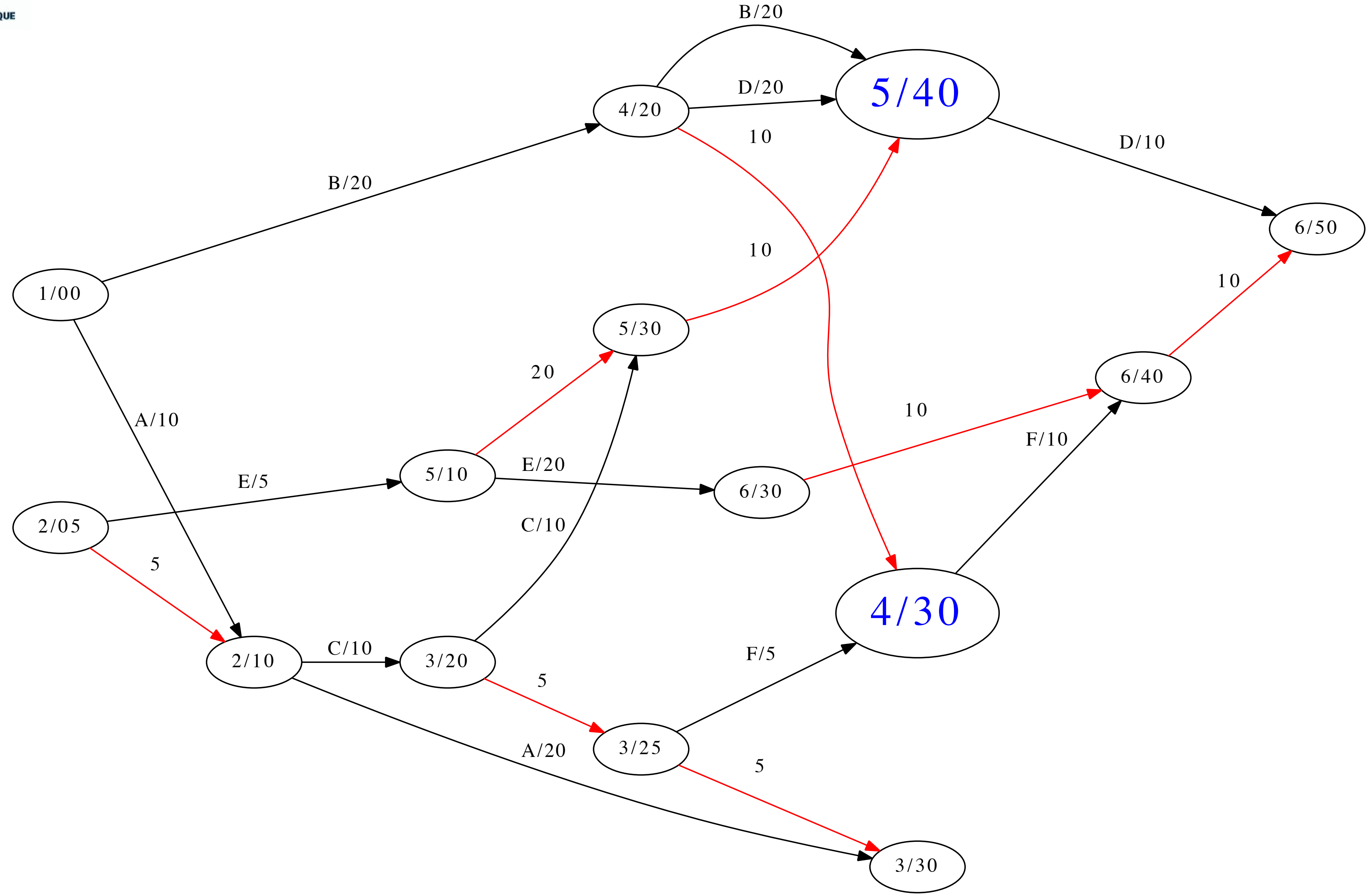
4/20	3/20	3/30

Finding a good itinerary



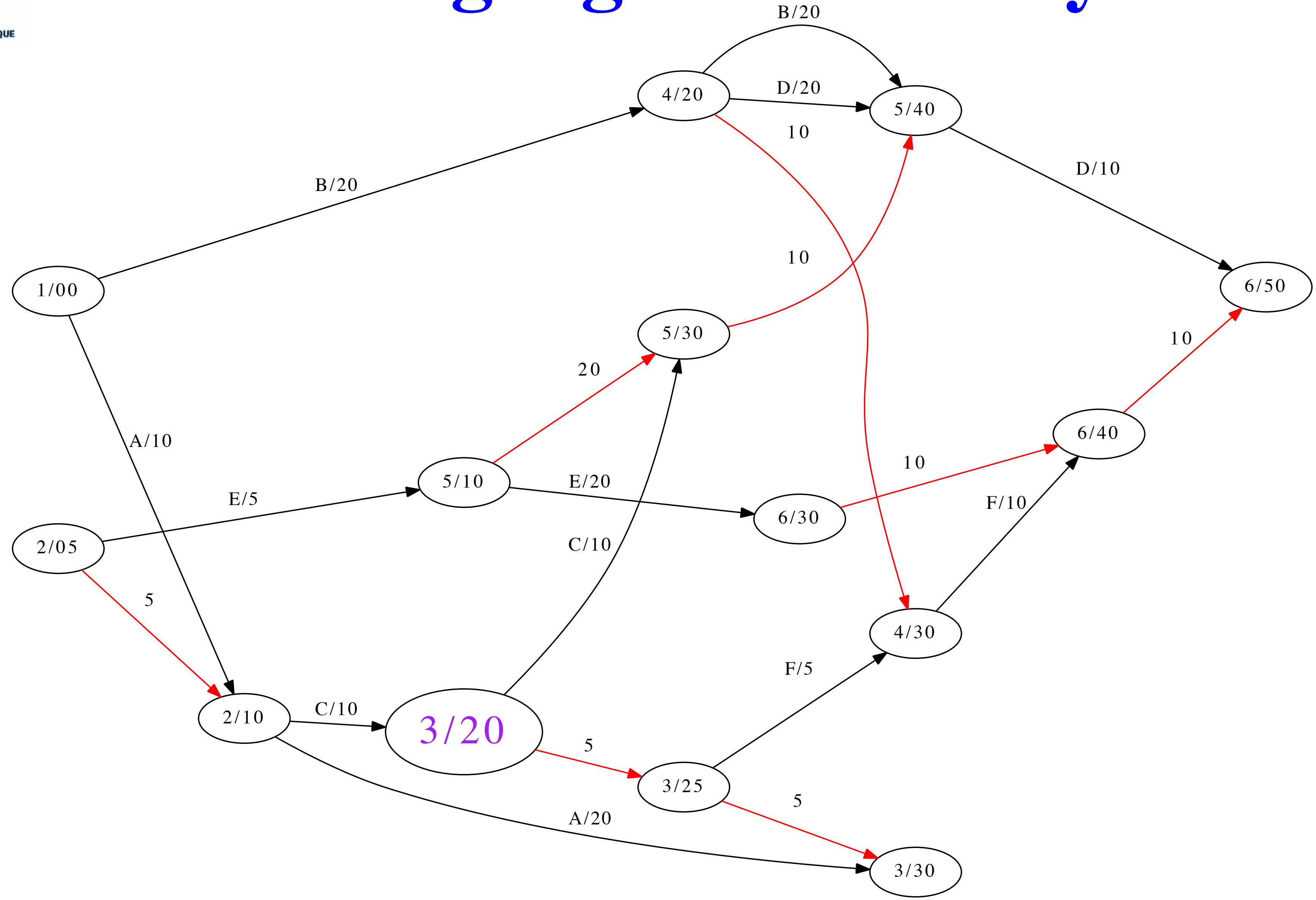
4/20 ← | 3/20 | 3/30 |

Finding a good itinerary



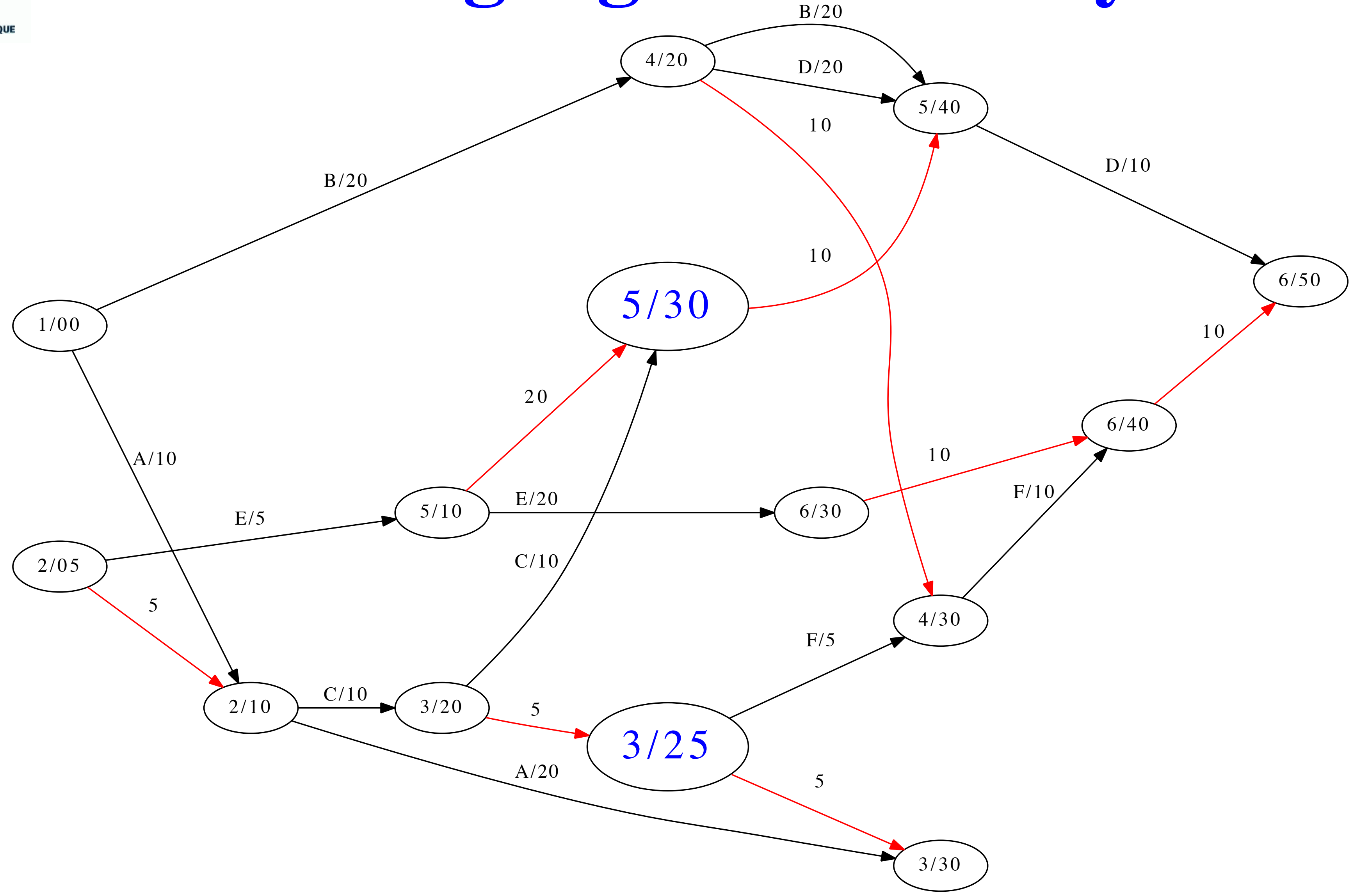
3/20	3/30	4/30	5/40

Finding a good itinerary



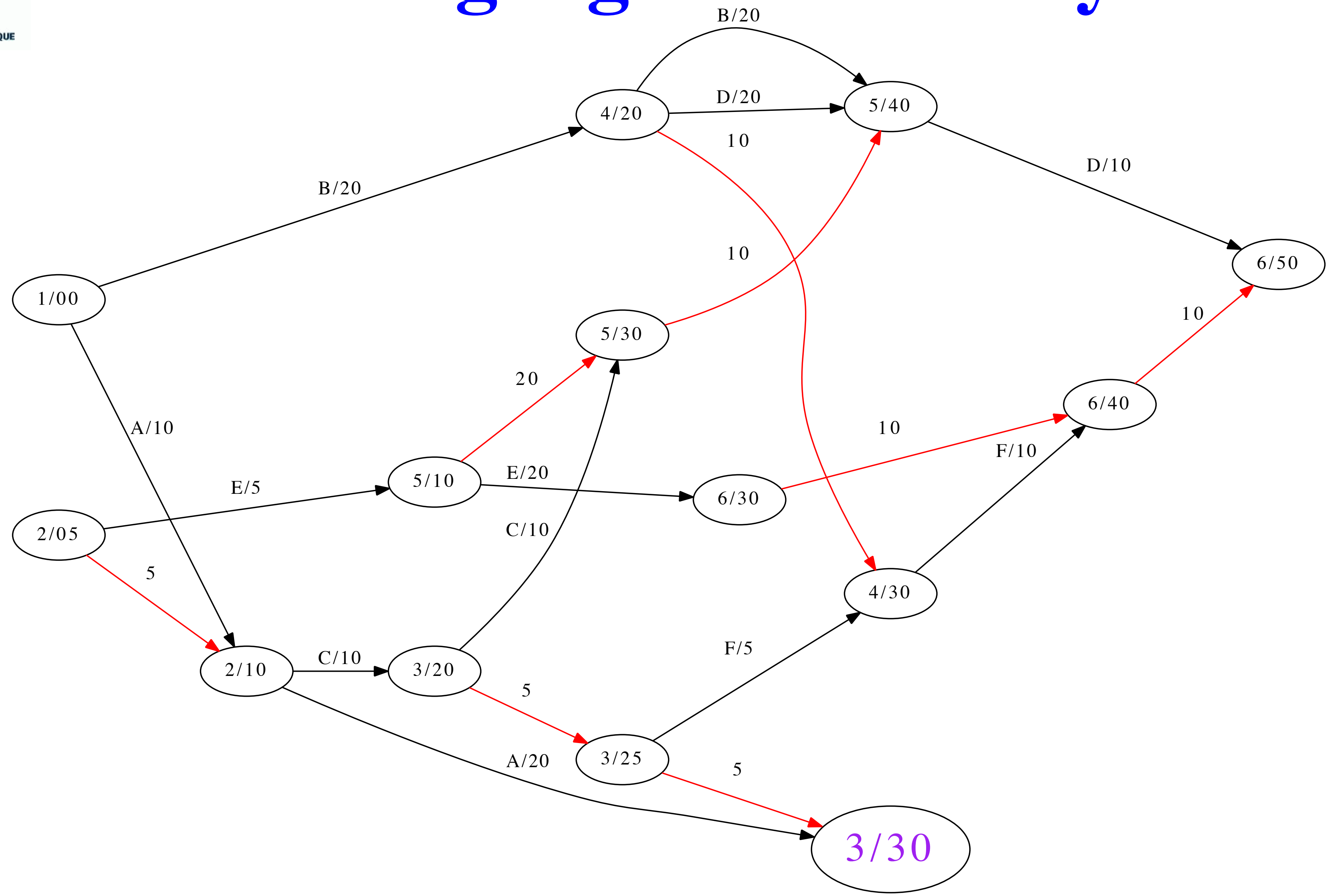
3/20 ← | 3/30 | 4/30 | 5/40 |

Finding a good itinerary



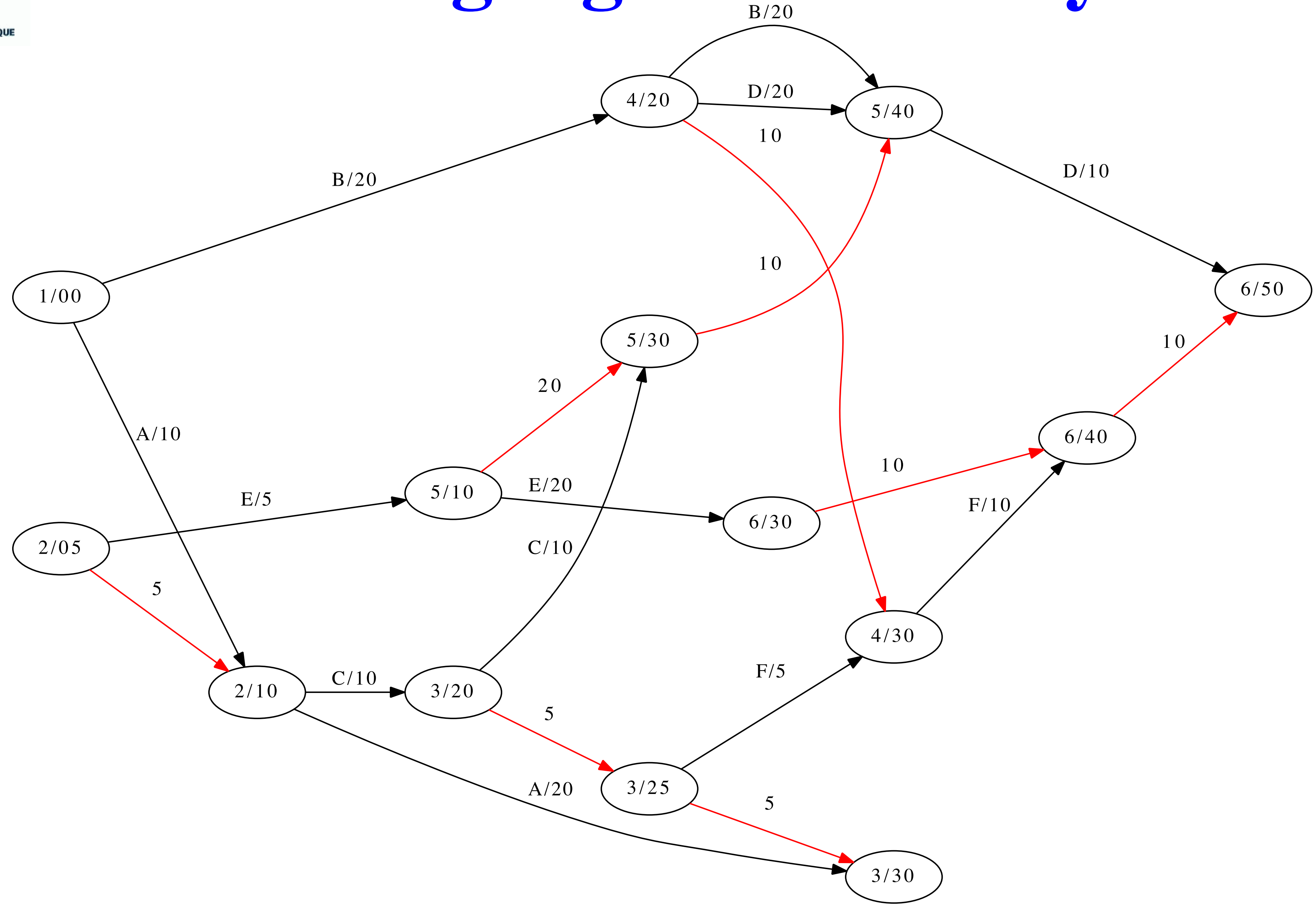
3/30	4/30	5/40	3/25	5/30

Finding a good itinerary



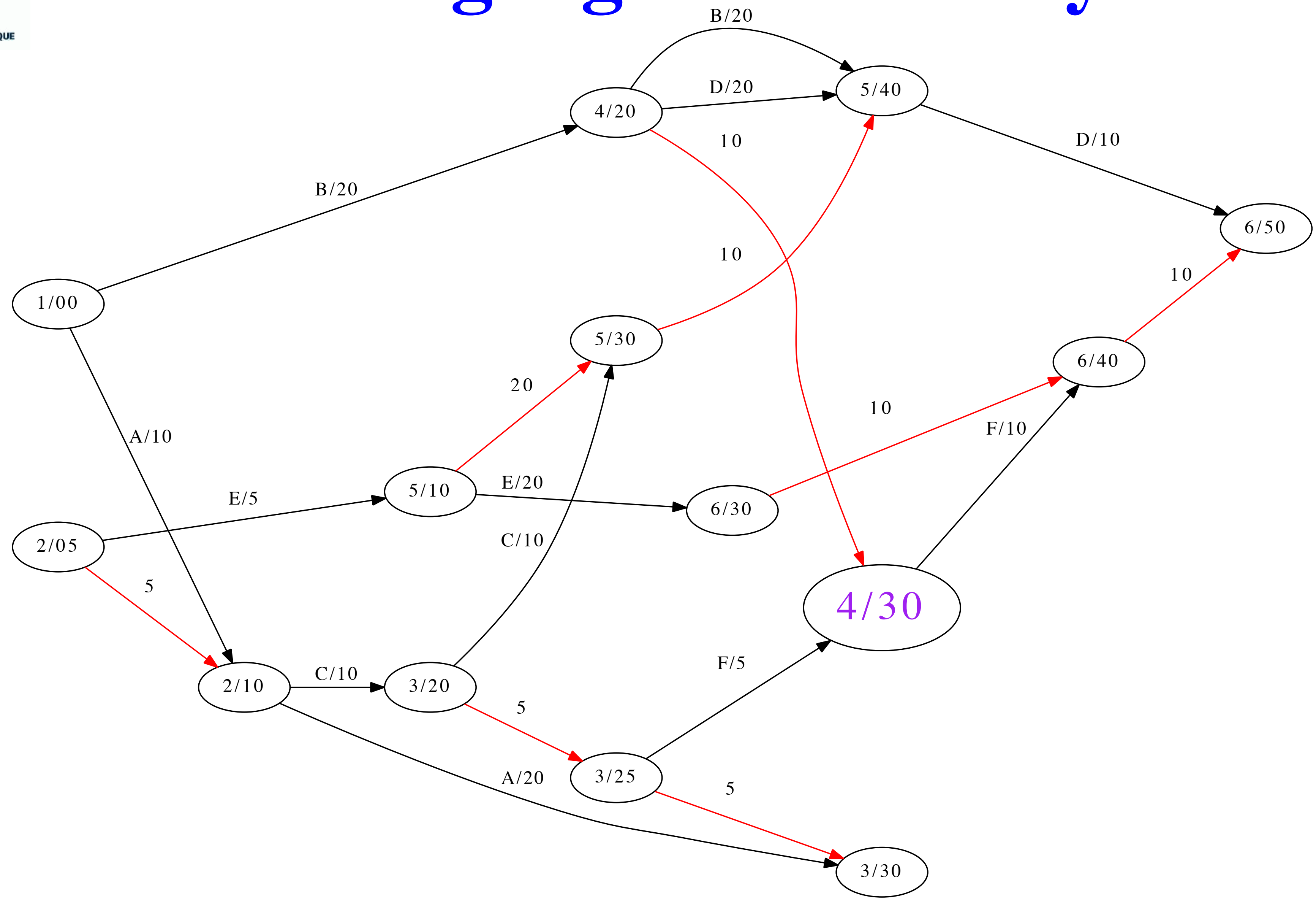
3/30 ← | 4/30 | 5/40 | 3/25 | 5/30 |

Finding a good itinerary



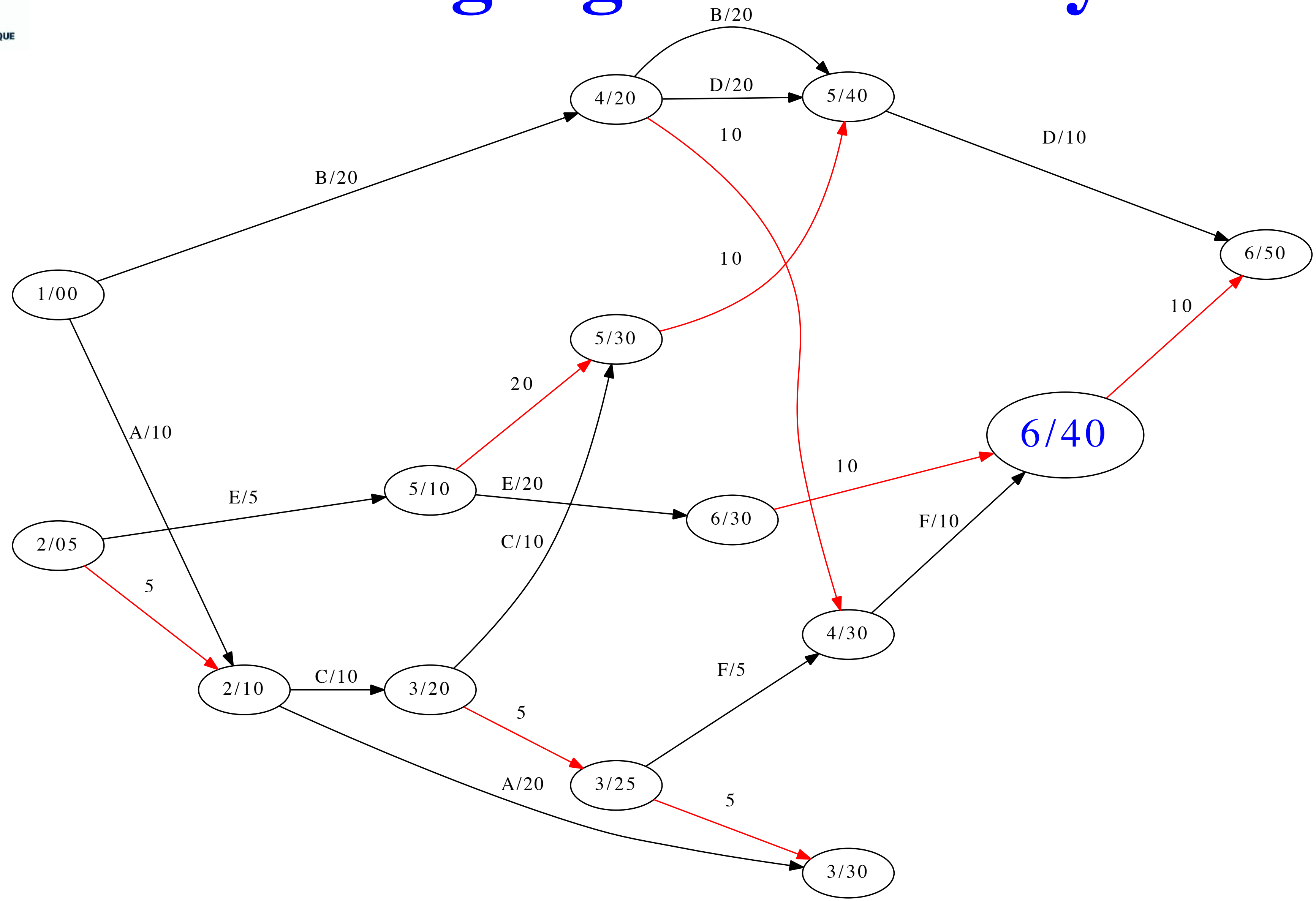
4/30	5/40	3/25	5/30

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

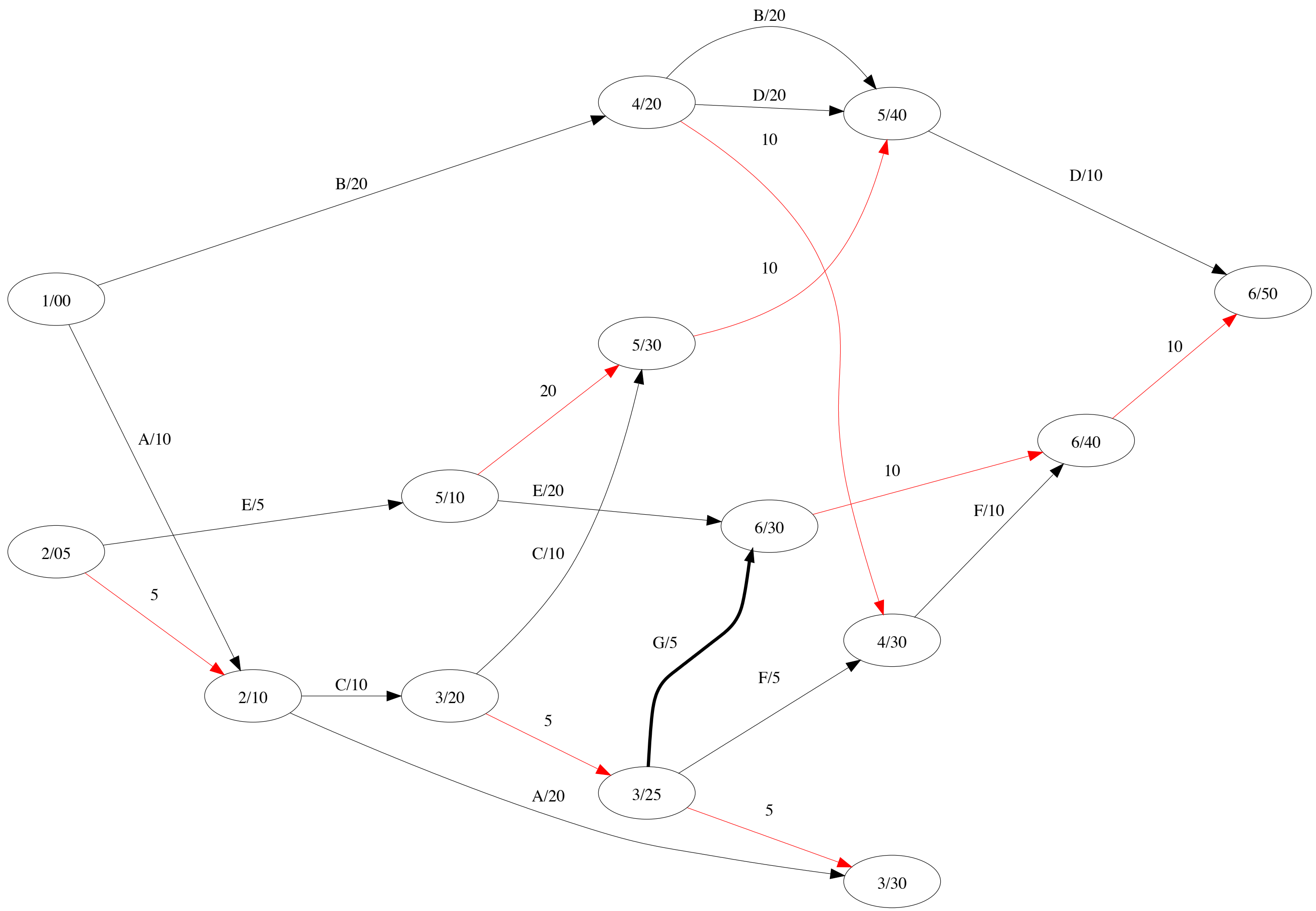
- Duration → actual path?
- *Store nodes out of queue with predecessors*

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

- Retrieve path backwards: 6/40→4/30→4/20→1/00

This ain't the fastest

Suppose there is a bus G with timetable 3/25 → 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?

- **Itinerary with fewest changes**
- *“bus, waiting, bus” counts as two changes, not one*
- Proof requires formalization (algorithm)
- Describe “queue” as a data structure

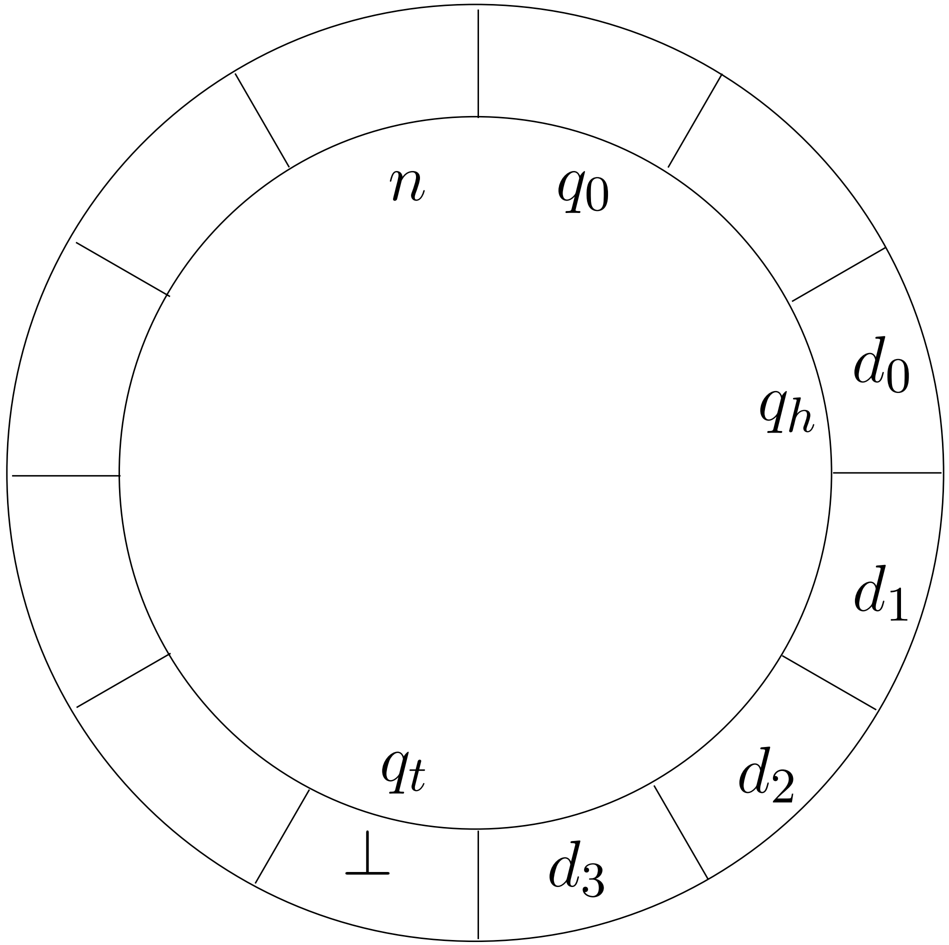
Queues

Queue operations

- Basic queue operations
  - `pushBack`: insert element at end of queue
  - `popFront`: retrieve and delete element at front of queue
  - `isEmpty`: is queue empty?
  - `size`: return queue size
- Need these operations to be $O(1)$

Circular arrays

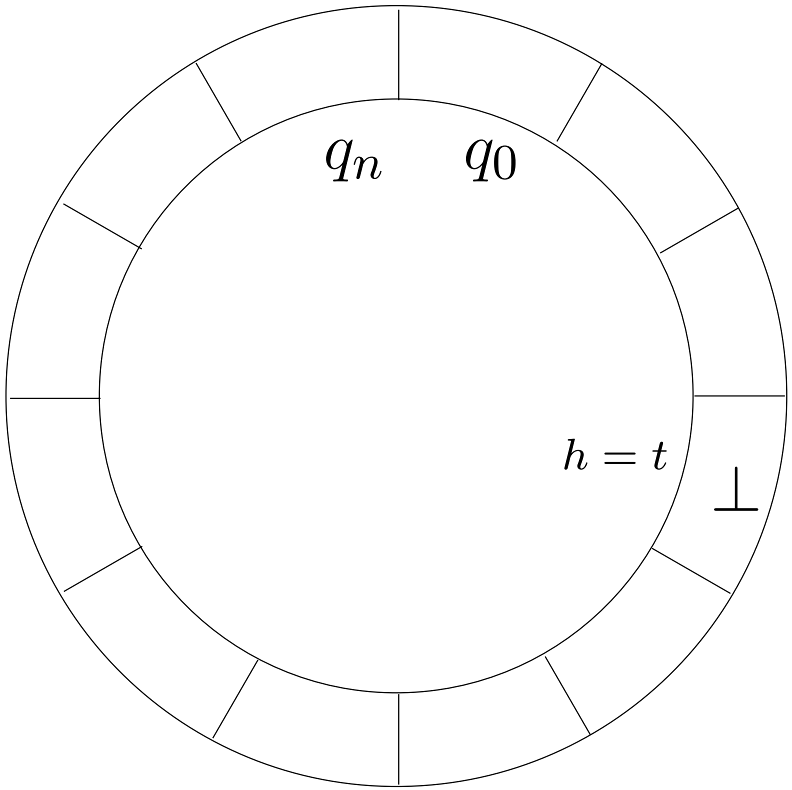
- Implementation using a *circular array* q [Mehlhorn & Sanders' book]
- Uses modular arithmetic (usually pretty fast)



- q_h: head of queue
- $q_t = \perp$ tail of queue
- q_i: unused for $0 \leq i < h$ and $t < i \leq n$
- circular array: array with modular index arithmetic

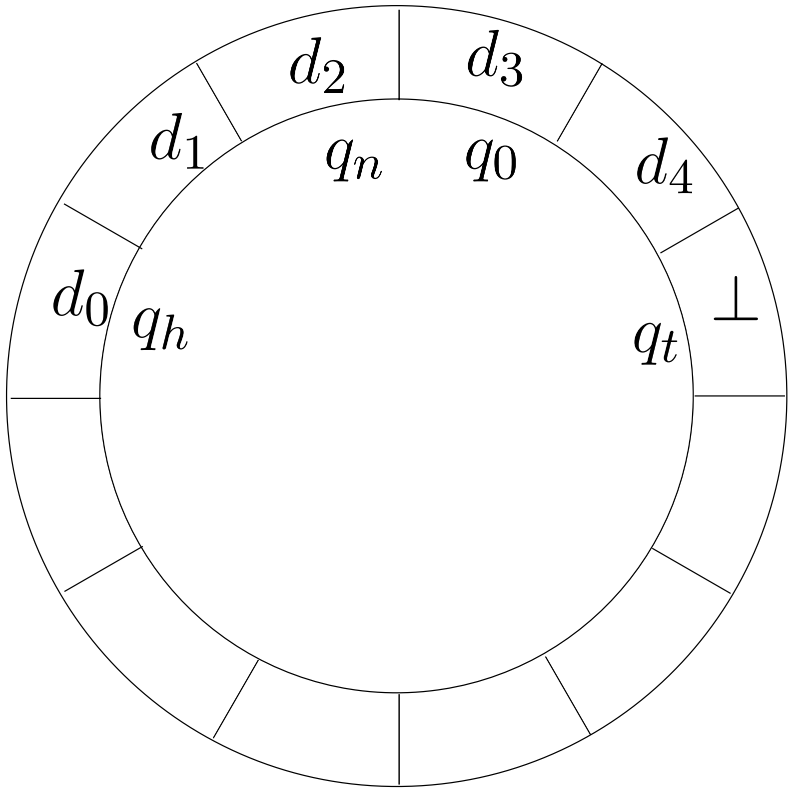
Size and emptiness

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



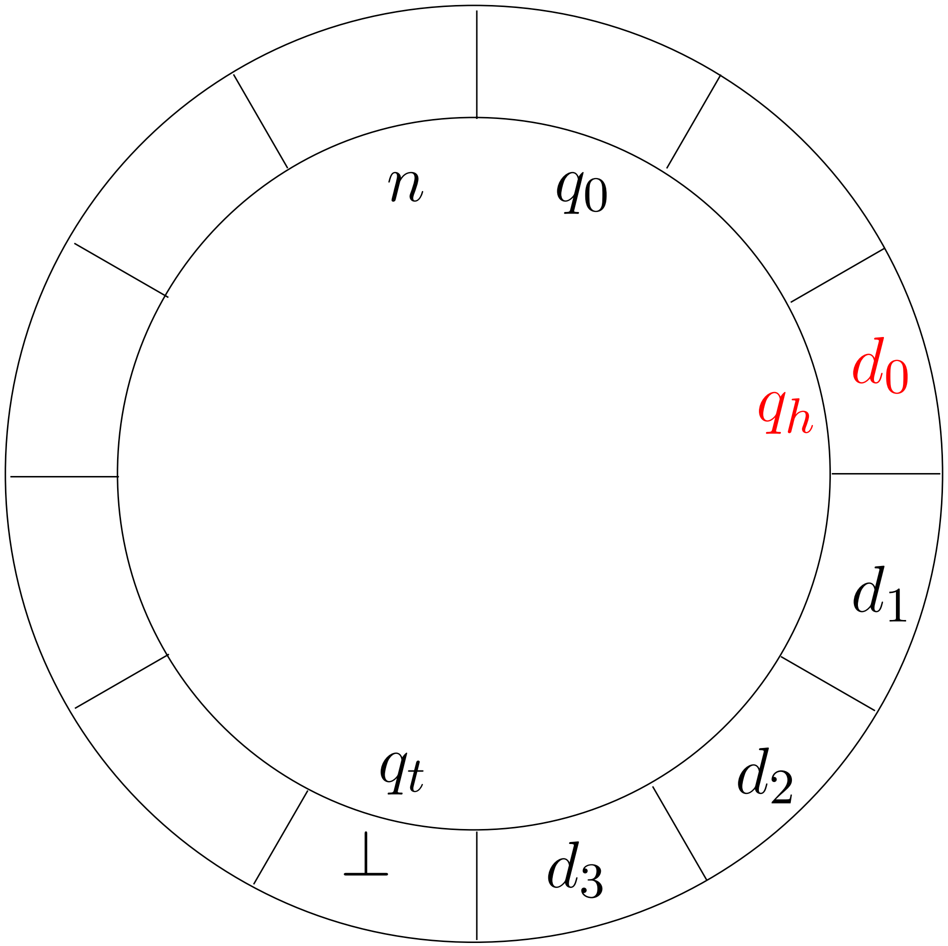
Size and emptiness

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



Read front of queue

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

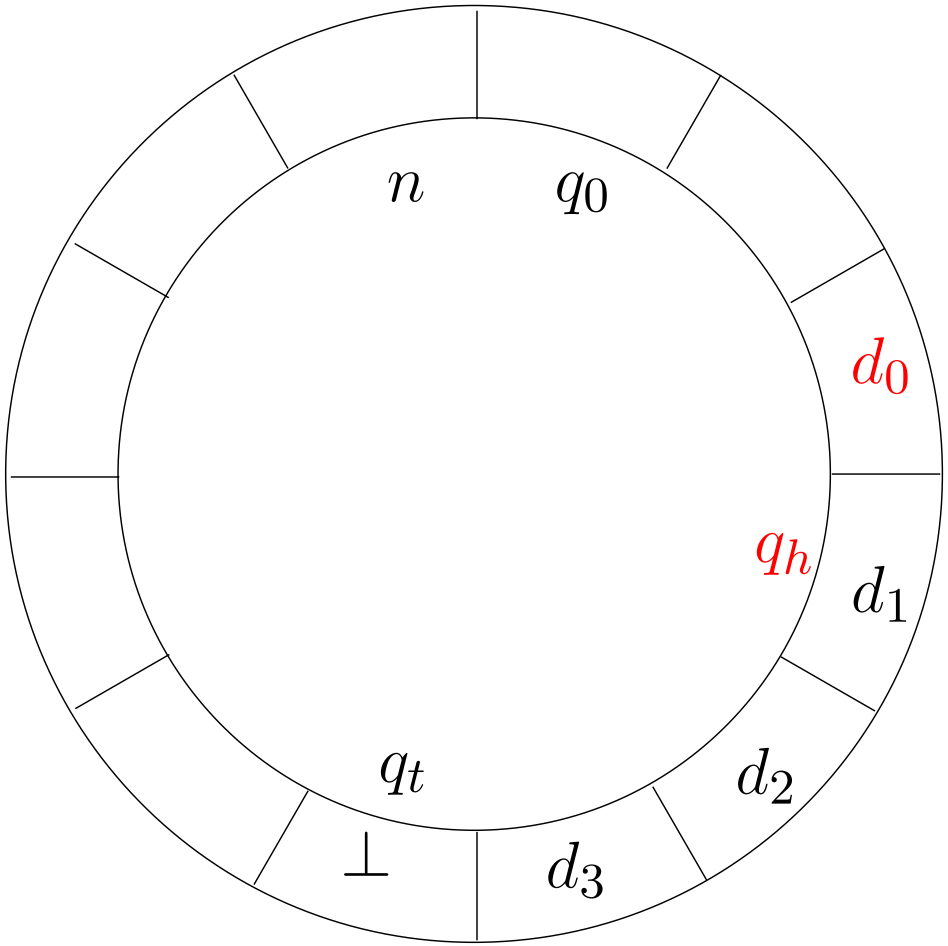


Read and delete front of queue

popFront() {

$p = q_h$;
$h = (h + 1) \bmod n$;
return p;

}



Insert at the end of queue

pushBack(d) {

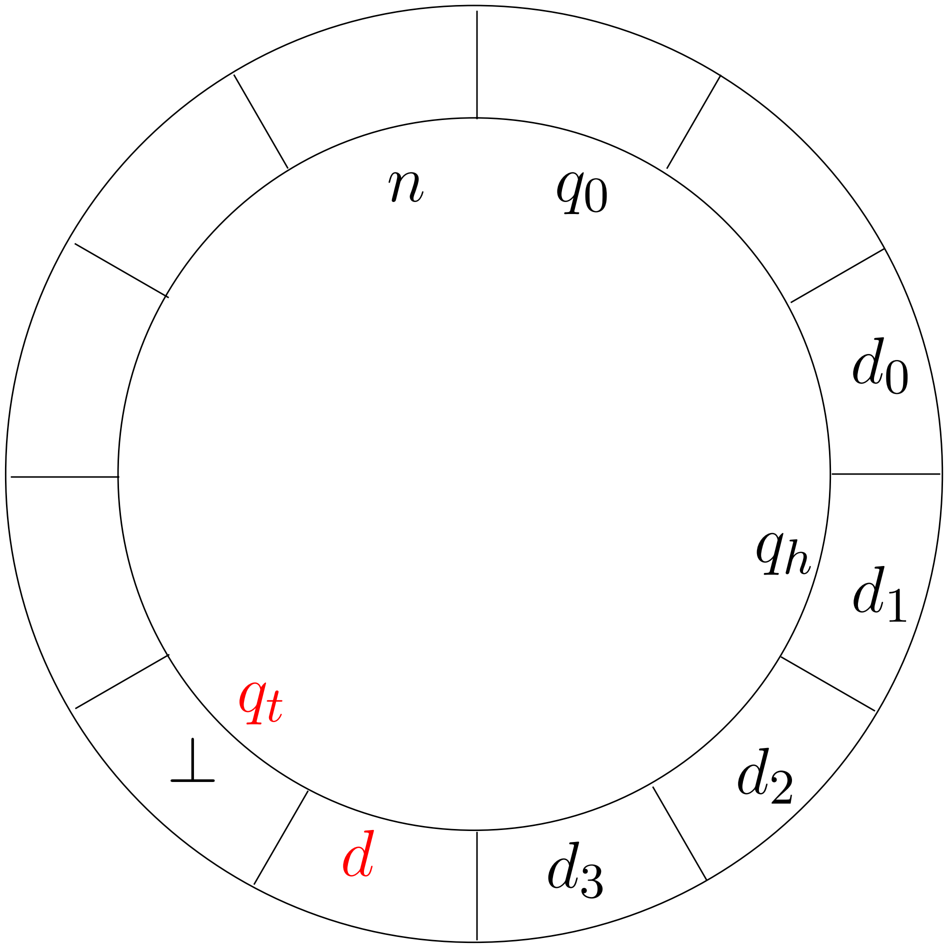
assert(size() $< n$)

$q_t = d$;

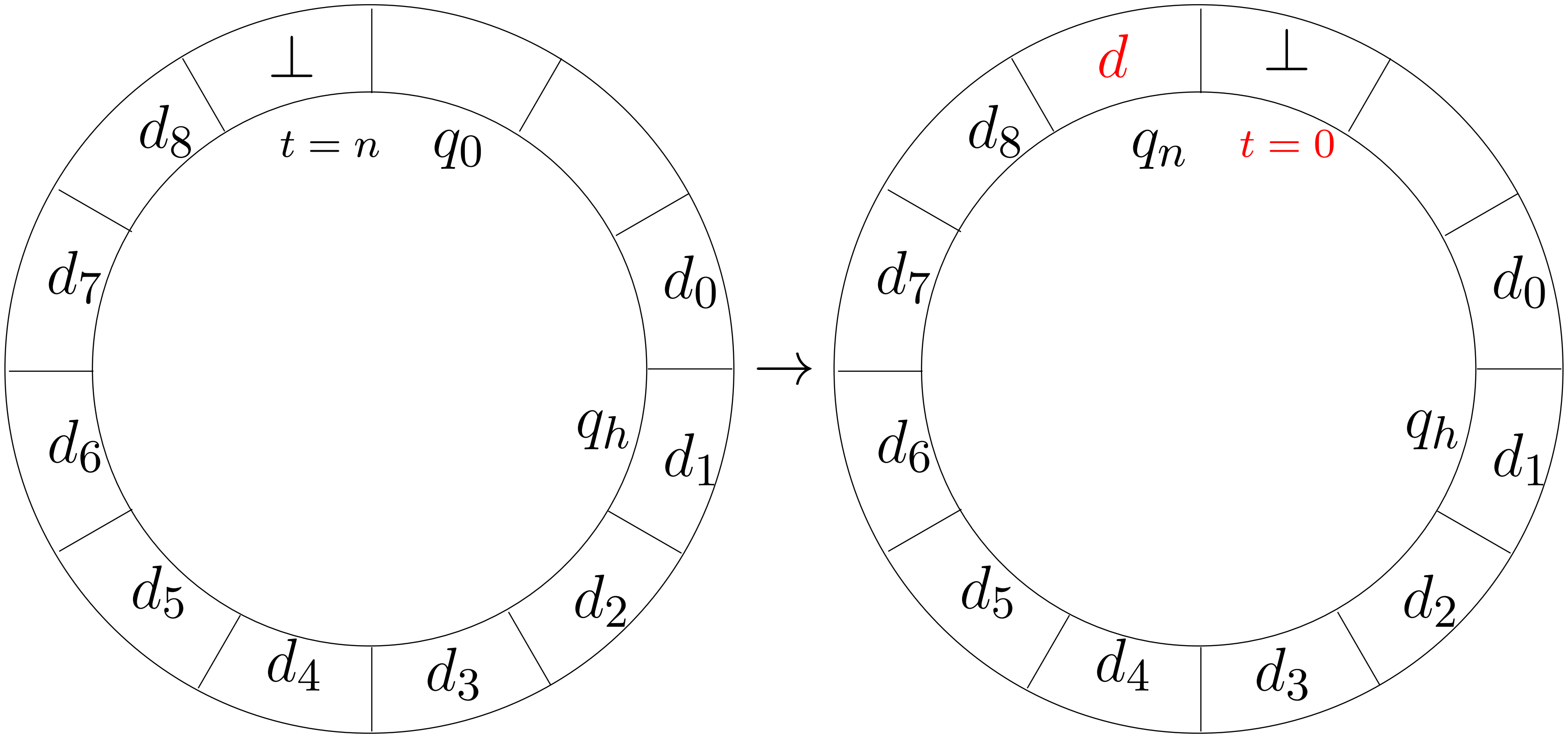
$t = (t+1) \bmod n$;

$q_t = \perp$;

}



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



$$(t = (t+1) \bmod n \wedge t = n-1) \Rightarrow t = 0$$

BFS

BFS: the idea

Explore nodes of a network starting from s

- start with $\texttt{pushBack}(s)$
- at any iteration,
  1. $u = \texttt{popFront}()$
  2. for each neighbour v of u,
  3. if v is the target, stop
  4. if v not already seen, $\texttt{pushBack}(v)$

The BFS algorithm

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

```
(Q, <) = {s}; R = {s};
while Q ≠ ∅ do
  u = min_< Q; Q ← Q ∖ {u};
  for v ∈ V (v ∼ u ∧ v ∉ R) do
    if v = t then
      return "t reachable";
    end if
    Q ← Q ∪ {v}, set v = max_< Q;
    R ← R ∪ {v};
  end for
end while
return "t unreachable";
```

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 $v \not\in R$ at Step 4, we have:
  Thm. 1

  No element of V enters Q more than once

A node hierarchy

- Consider function $\alpha : V \to \mathbb{N}$:

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

  at Step 8, let $\alpha(v) = \alpha(u) + 1$

- Ranks $v \in V$ by distance from s in terms of "arrows"
- E.g. if $s \to u$, then u's distance from s is 1
  if $s \to u \to v$, v's distance from s is 2

The BFS, again

```
(Q,<) = {s}; R = {s};
α(s) = 0;
while Q ≠ ∅ do
  u = min_< Q; Q ← Q ∖ {u};
  for v ∈ V (v ∼ u ∧ v ∉ R) do
    α(v) = α(u) + 1;
    if v = t then
      return "t reachable";
    end if
    Q ← Q ∪ {v}, set v = max_< Q;
    R ← R ∪ {v};
  end for
end while
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, $\alpha(v_k) = k$

Thm. 3

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

Thm. 4

No itinerary found by BFS has repeated elements

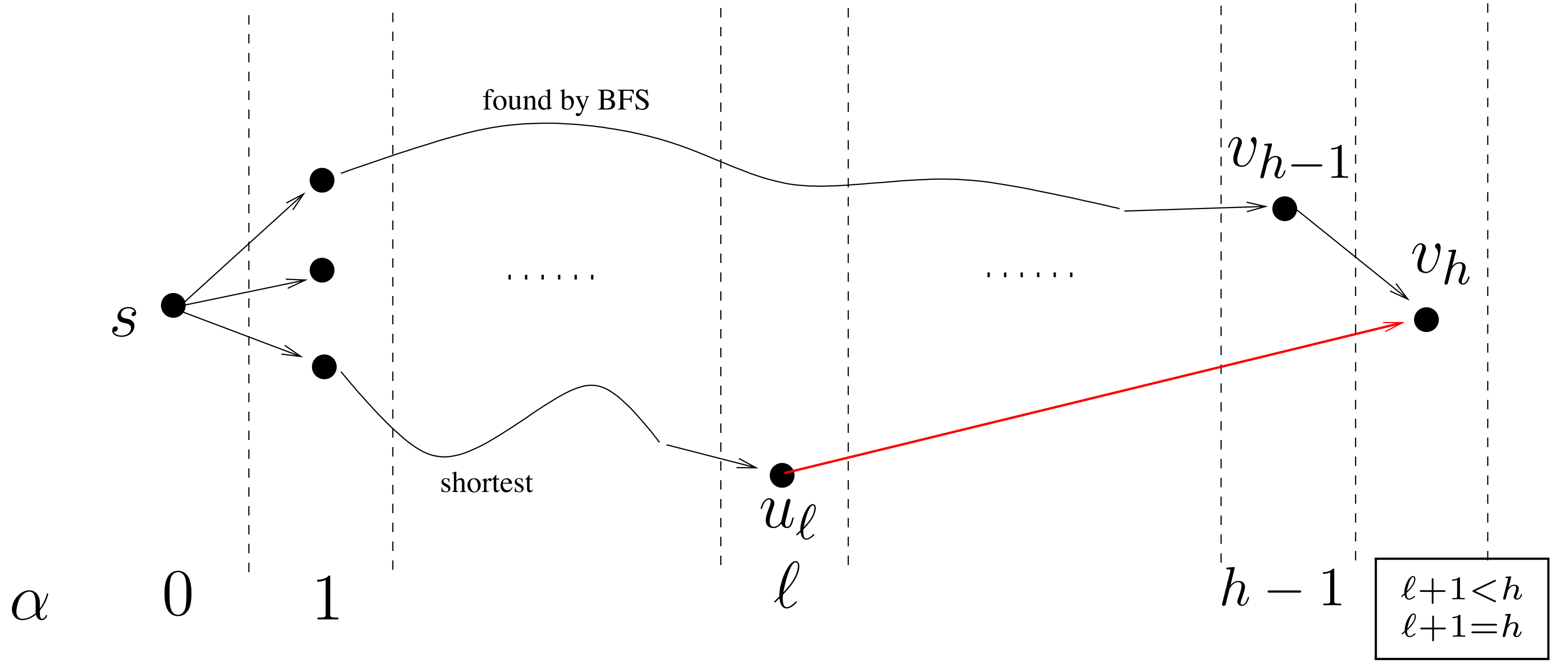
Thm. 5

The function α is well defined

Fewest changes

- Aim to prove that **BFS finds an itinerary with fewest changes**
- **Remark**: #changes in an itinerary = #nodes/edges
  Thm.
  BFS finds a path with fewest edges

Idea of proof:
by contradiction



Finding all shortest itineraries

- Delete Steps 7-9
- All elements in V enter and exit Q
- Finds shortest itineraries from s to all elements of V

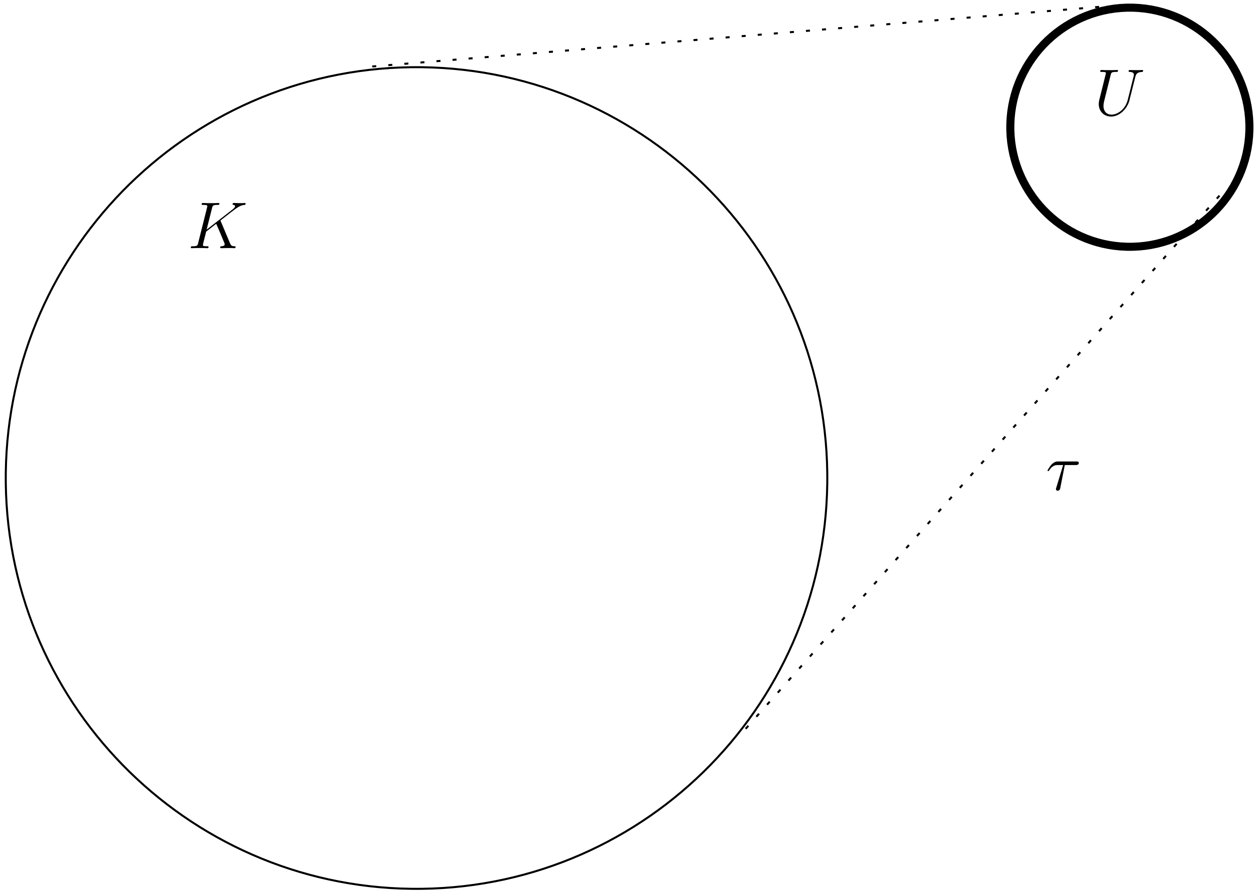
WARNING: BFS will *not* find shortest paths in a weighted graph unless all the arc costs are 1

Hashing

Motivating example

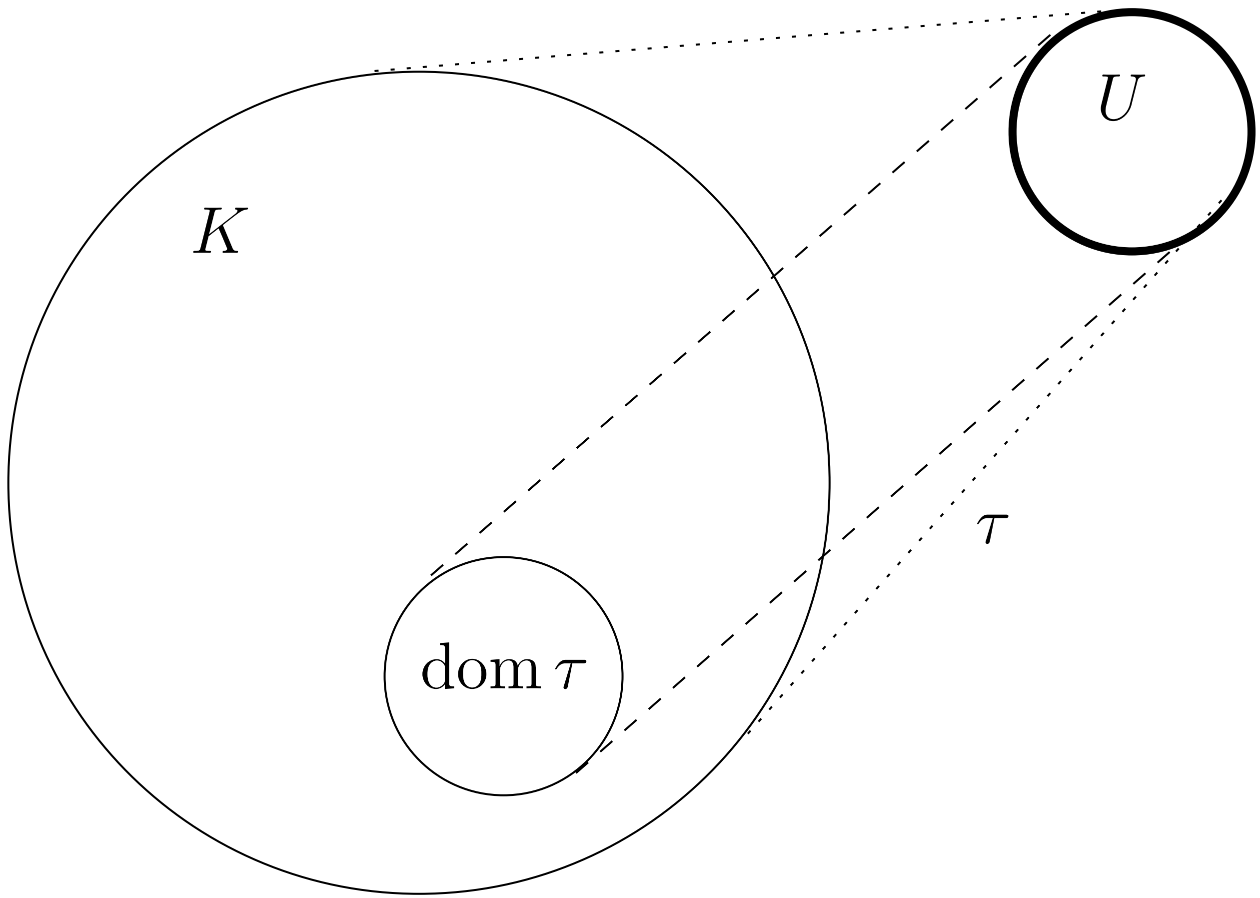
- The phonebook with n entries
- Each page corresponds to a character
- Page with character k contains all names beginning with k
- Easy to search:
  - 26 chars in alphabet: $O(1)$
  - L lines per page, L does not depend on n: $O(1)$
- Search is $O(1)$

The idea



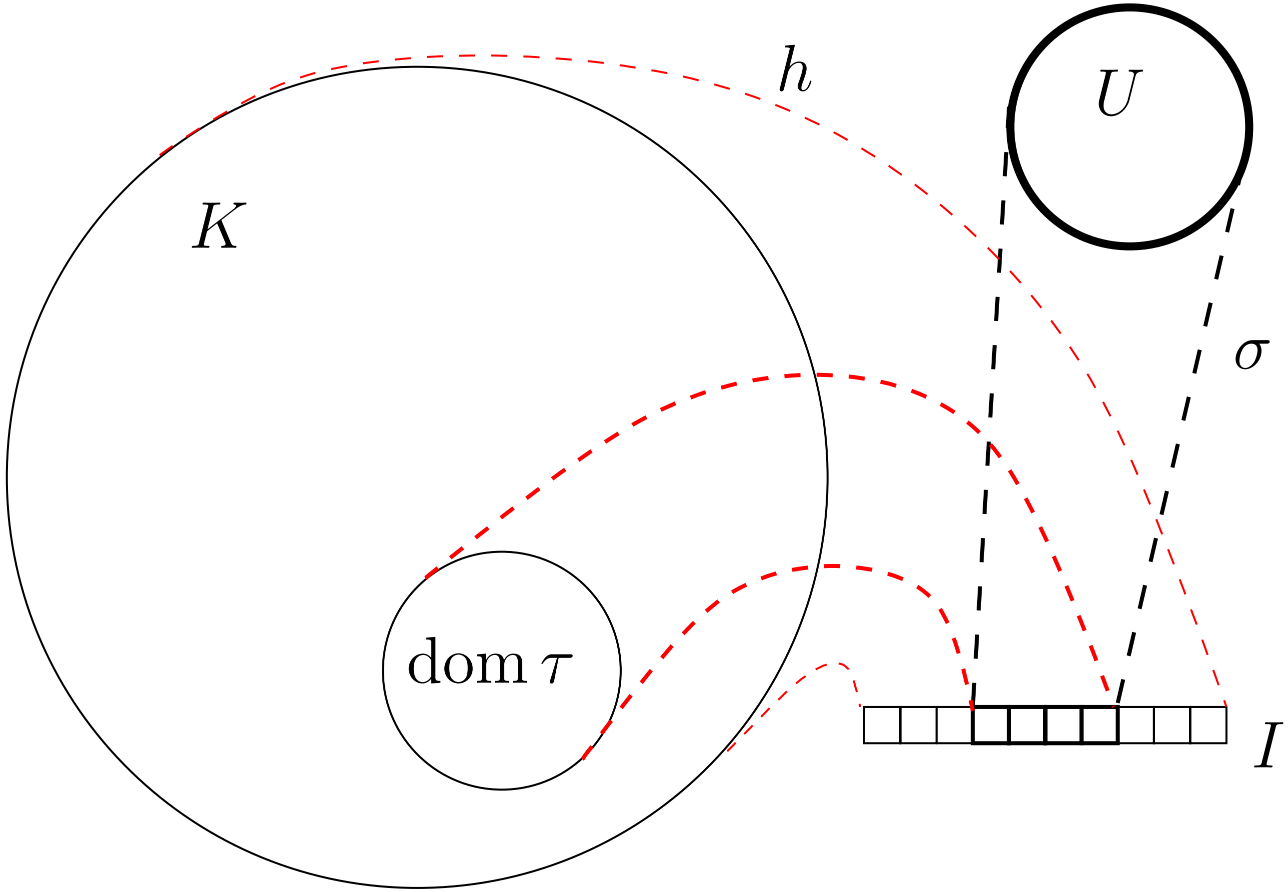
- K a very large set of keys; U: a set of objects; $\tau : K \to U$: a table
- Assume K too large to store, but $\mathrm{dom}\,\tau$ is small
- Find a function $h : K \to I$ with $I = \{0, 1, \ldots, p-1\}$ and $|I| \approx |U|$, then store $u = \tau(k)$ at $\sigma(i)$ where $i = h(k)$

The idea



- K a very large set of keys; U: a set of objects; $\tau : K \to U$: a table
- Assume K too large to store, but $\operatorname{dom}\tau$ is small
- Find a function $h : K \to I$ with $I = \{0, 1, \ldots, p-1\}$ and $|I| \approx |U|$, then store $u = \tau(k)$ at $\sigma(i)$ where $i = h(k)$

The idea



- K a very large set of keys; U: a set of objects; $\tau : K \to U$: a table
- Assume K too large to store, but $\operatorname{dom}\tau$ is small
- Find a function $h : K \to I$ with $I = \{0, 1, \ldots, p-1\}$ and $|I| \approx |U|$, then store $u = \tau(k)$ in array element $\sigma(i)$ where $i = h(k)$

Why not a list?

- Consider list of pairs $(\text{key}, \text{record})$
- Finding is $O(n)$
- Time-inefficient

Why not an array?

- Consider array of records indexed by keys
- Finding is $O(1)$
- Suppose keys are $\{1, 16, 1643, 1094382\}$
- Need to allocate space for 1094382 records, just need 4
- Space-inefficient

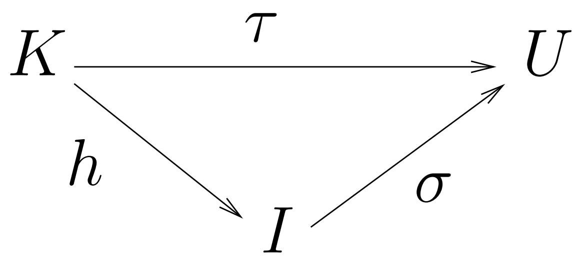
Problem setting

- K =*keys*, U =*records*
- Associate some keys with records
- Get an injective *table function* $\tau : K \to U$, with $\mathrm{dom}\,\tau \subsetneq K$
- **Problem**:

  Given a key $k \in K$, determine whether $k \in \mathrm{dom}\,\tau$

Hash tables

- Consider **index set** I s.t. $|I| \approx |\operatorname{dom}\tau| \ll |K|$
- **Hash table**: function $\sigma : I \to U$
- **Hash function**: function $h : K \to I$ s.t. $\tau = \sigma \circ h$



- $\Rightarrow$ Store u in σ at position $h(k)$
- Get $\sigma(h(k)) = \tau(k) = u$

Collisions

- By above, $k \in \operatorname{dom} \tau \Leftrightarrow h(k) \in I$
- Scheme only works if h is injective
- If not, get *collisions* (see phonebook)
- If collisions, let $\sigma(h(k)) =$ all u's with equal $h(k)$

Last nagging doubt

- Need to store $h : K \to I$ somewhere
- List is time-inefficient
- Array is space-inefficient
- Are we simply shifting the problem?

The magic

No need to store h explicitly

- Define $h(k)$ using a “short description”
- A formula applied to the description of k
- E.g. phonebook:
  - let $k =$ Leo
  - ASCII code: L $= 76$, e $= 101$, o $= 111$
  - $h(k) = 76 + 101 + 111 = 288$
    collisions, $h($HHHH$) = 288$ too
  - $h(k) = 76 \times 113^2 + 101 \times 113 + 111 = 981968$
    no collisions

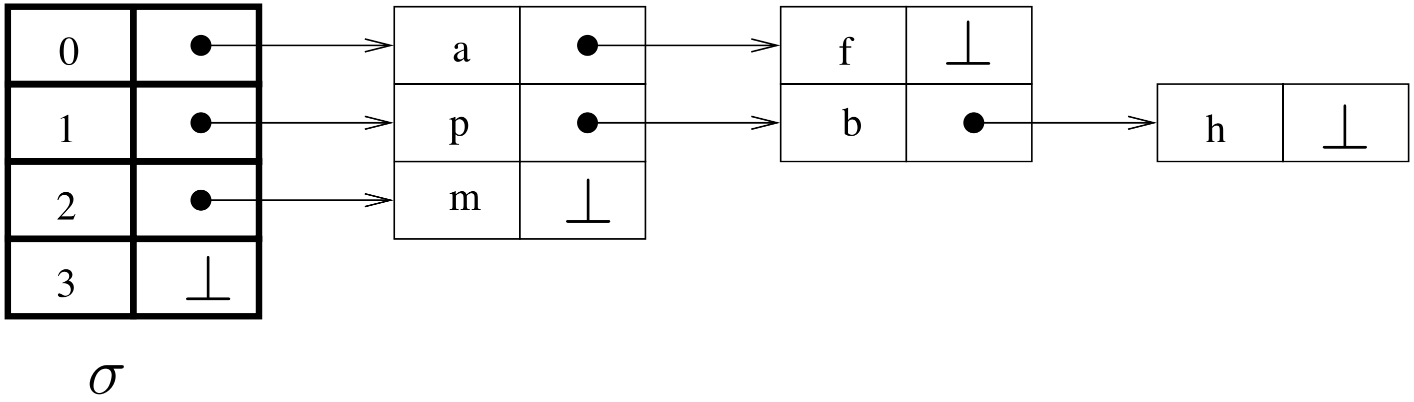
Collisions are likely

- Most functions are *not* injective
- $|I|^{|K|}$ functions from $K \to I$
- If $|I| < |K|$, none is injective
- If $|I| \geq |K|$:
  - $|I|$ ways to choose the image of the first element of K,
  - $|I| - 1$ ways to choose the second, and so on
  - get $\begin{pmatrix} |I| \\ |K| \end{pmatrix}$ injective functions $K \to I$
- If $|K| = 31$ and $|I| = 41$, there are around 10^{50} functions, only 10^{43} of which are injective (*one in ten million: rare*)

This calculation by D. Knuth

Chaining

- Deal with collisions
- Store all records with same hash k in a list
- Store list at $h(k)$
- Example:



$h(\mathtt{a}) = h(\mathtt{f}) = 0$
$h(\mathtt{p}) = h(\mathtt{b}) = h(\mathtt{h}) = 1$
$h(\mathtt{m}) = 2$
$\perp$: not_found

Finding with collisions

- Finding in a hash table with collisions:

  if $|\sigma(h(k))| = 0$ **then**
  return not_found
  else if $|\sigma(h(k))| = 1$ **then**
  return $\sigma(h(k))$
  else
  return $\sigma(h(k))$.find(k)
  end if

- Reduce collisions:

  Injective or “almost injective” hash functions

Good hash functions

- In general, consider data as number sequences $(k_1, \dots, k_\ell)$
- Consider any number sequence $a = (a_1, \dots, a_k)$
- Let p be the smallest prime $\geq |U|$
- The following hash function family is almost injective

$$h_a(k) = \sum_{j \leq \ell} a_j k_j \pmod{p}$$

- Choice of a can make a difference

Complexity: worst-case

- **Assume**:
  - length of key k is $O(1)$ w.r.t. $n = |\tau|$
  - hash function evaluate in $O(1)$ w.r.t. n
- **Worst case**:
  - $\exists i \in I \; \forall k \in K \quad h(k) = i$
  - all keys stored in same sequence $\sigma(i)$: get $O(n)$

Complexity: average case

- **Assume**:
  - probability that $h(k) = h(k')$ for $k \neq k'$: $\frac{1}{|I|}$
  - this probability is independent of k, k'
- L_k: random variable for $|\sigma(h(k))|$
- Scanning $\sigma(h(k))$: $O(\mathsf{E}(L_k))$
- $X_{k\ell}$: random indicator variable, $X_{k\ell} = 1$ if $h(k) = h(\ell)$, 0 othw.

$$\begin{aligned} L_k &= \sum_{u \in \mathrm{ran}\,\tau} X_{k\ell} \\ \mathsf{E}(L_k) &= \sum_{u \in \mathrm{ran}\,\tau} \mathsf{E}(X_{k\ell}) \\ &= \sum_{u \in \mathrm{ran}\,\tau} \frac{1}{|I|} = \frac{|\tau|}{|I|} = \alpha \end{aligned}$$

Find, insert, remove in $O(1+\alpha)$

Application: comparing objects

- Objects can occupy lots of memory
- How to test $a = b$ efficiently?
- Byte comparison: $O(\min(|a|, |b|))$, inefficient
- Test `a.hashCode()==b.hashCode()` instead, $O(1)$
- Java's `hashCode()` function is good
- Small chance of collisions
- ...but chance nonetheless!
- Could have equal hashcodes but different objects
- If hashcodes are different, objects are different

Application: making $$

- Finding good hash functions is hard
- Requires lots of CPU time
- This computer work is worth some money

  `http://bitcoin.org/`

- Moreover, it prevents spam

  `http://hashcash.org/`

End of lecture 2