QEEE DSA05
DATA STRUCTURES AND ALGORITHMS

G VENKATESH AND MADHAVAN MUKUND

LECTURE 8, 2 SEPTEMBER 2014

Tasks with constraints

- For a foreign trip you need to
  - Get a passport
  - Buy a ticket
  - Get a visa
  - Buy travel insurance
  - Buy foreign exchange
  - Buy gifts for your hosts

Tasks with constraints

- There are constraints
  - Without a passport, you cannot buy a ticket or travel insurance
  - You need a ticket and insurance for the visa
  - You need the visa for foreign exchange
  - You don’t want to invest in gifts unless the trip is confirmed

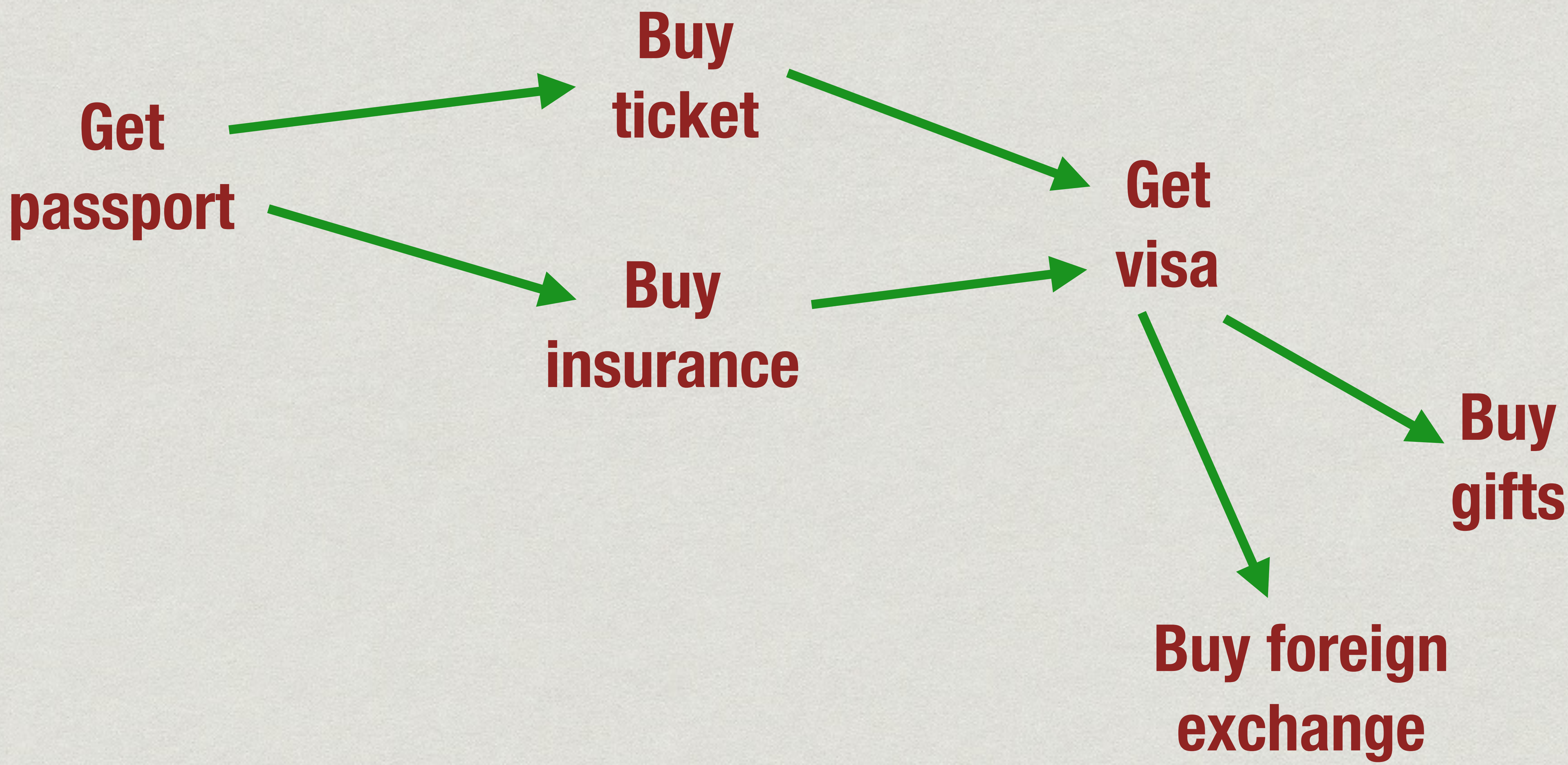
Goal

- Find a sequence in which to complete the tasks, respecting the constraints

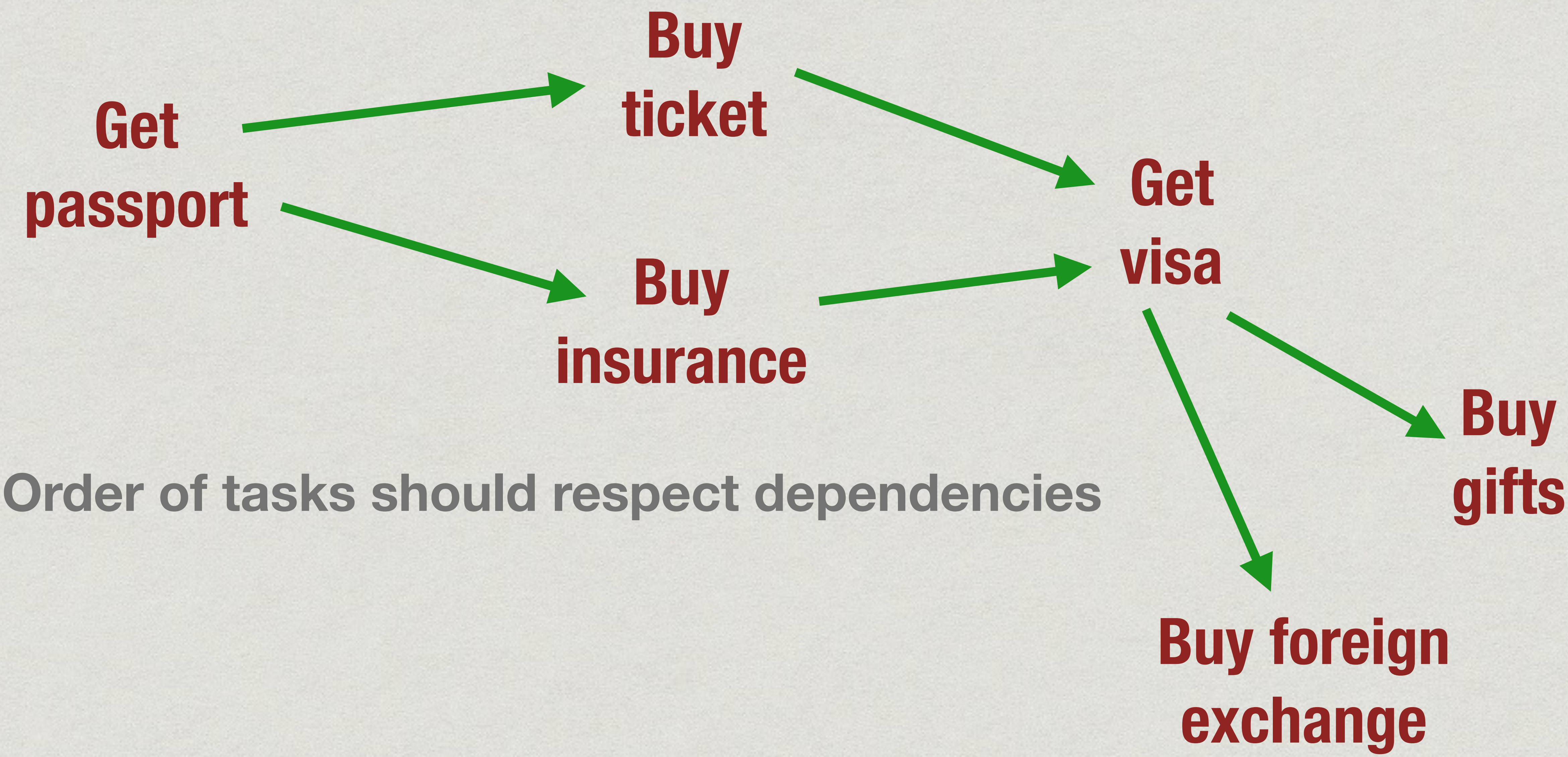
Model using graphs

- Vertices are tasks
- Edge from Task1 to Task2 if Task1 must come before Task2
  - Getting a passport must precede buying a ticket
  - Getting a visa must precede buying foreign exchange

Our example as a graph

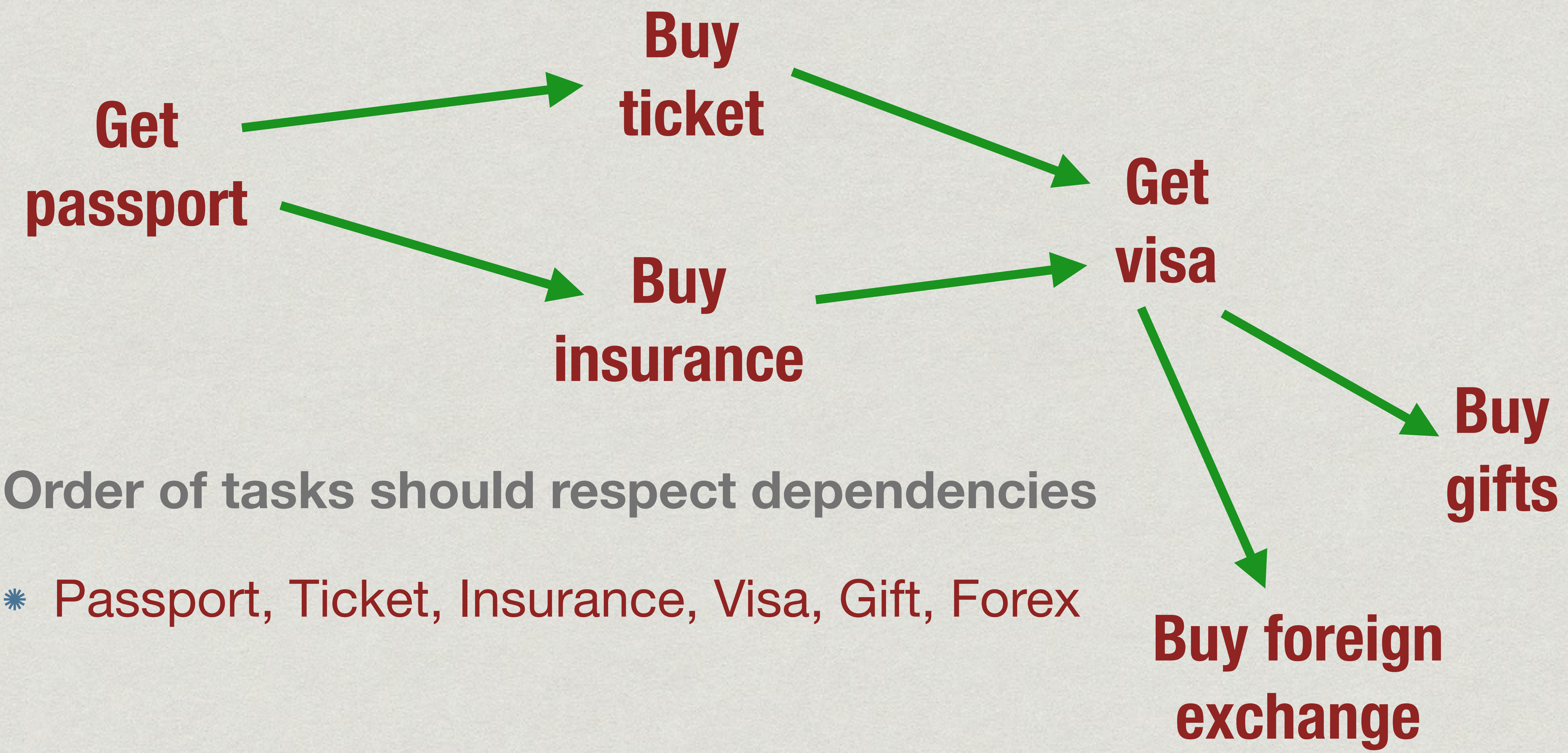


Our example as a graph



Order of tasks should respect dependencies

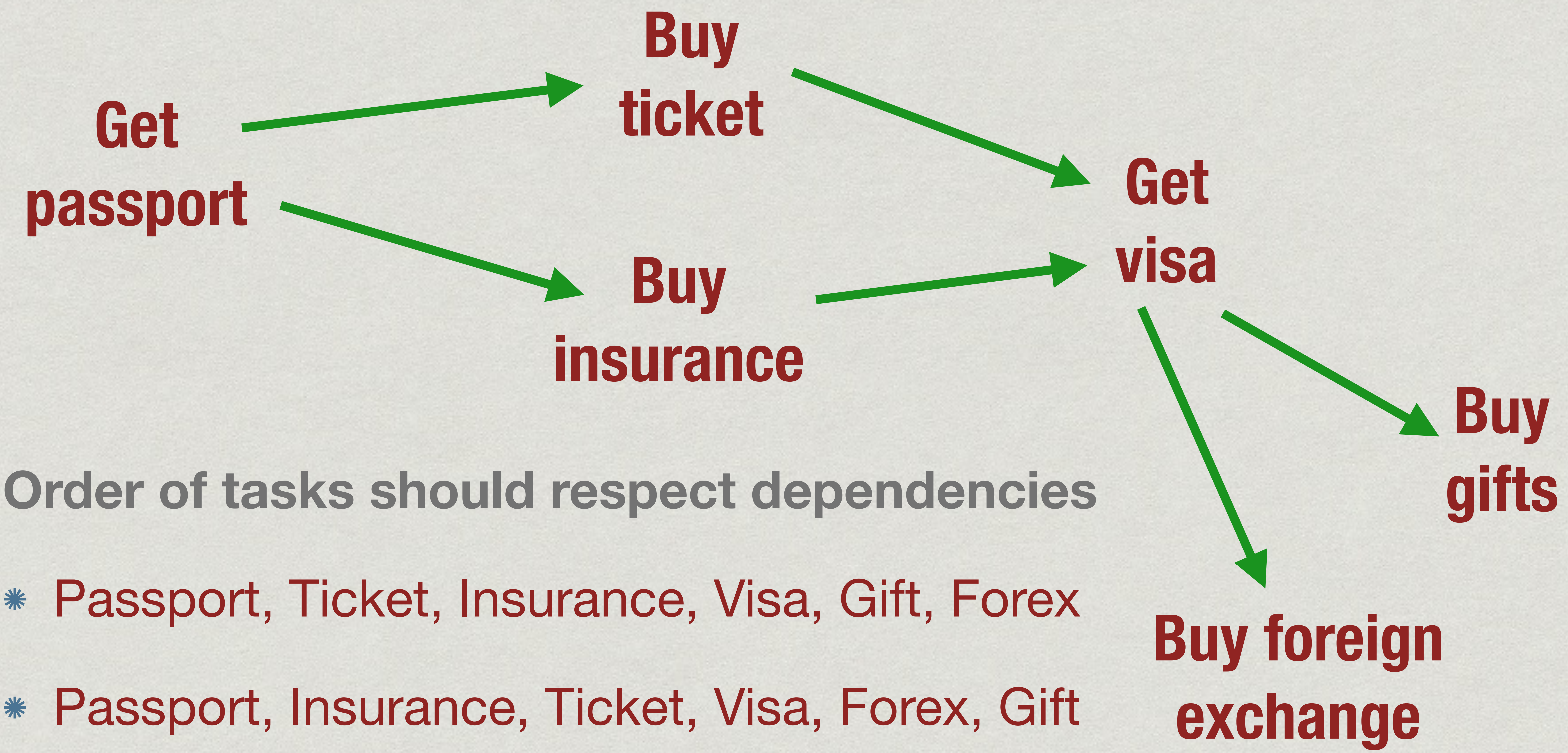
Our example as a graph



Order of tasks should respect dependencies

- Passport, Ticket, Insurance, Visa, Gift, Forex

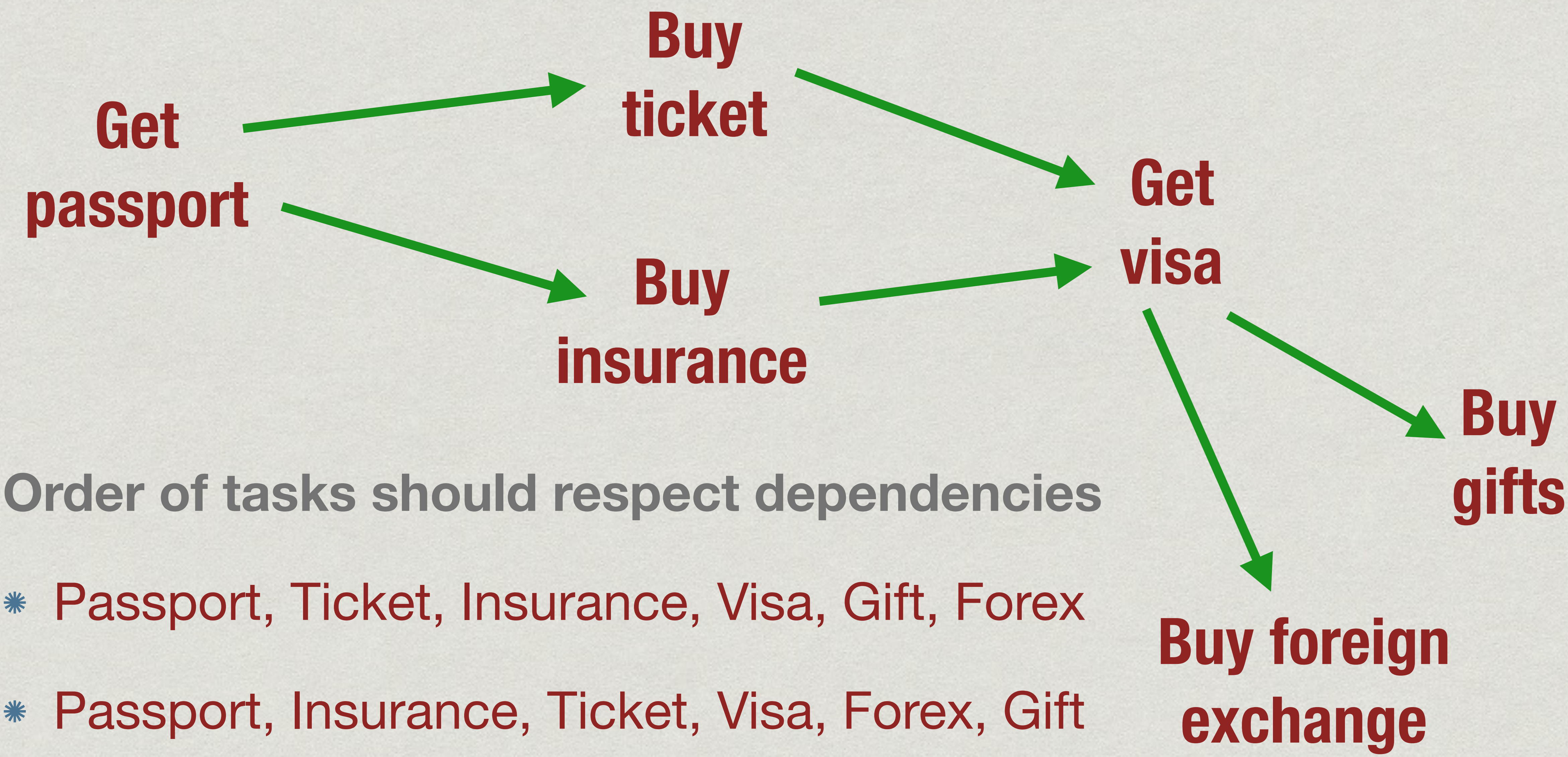
Our example as a graph



Order of tasks should respect dependencies

- Passport, Ticket, Insurance, Visa, Gift, Forex
- Passport, Insurance, Ticket, Visa, Forex, Gift

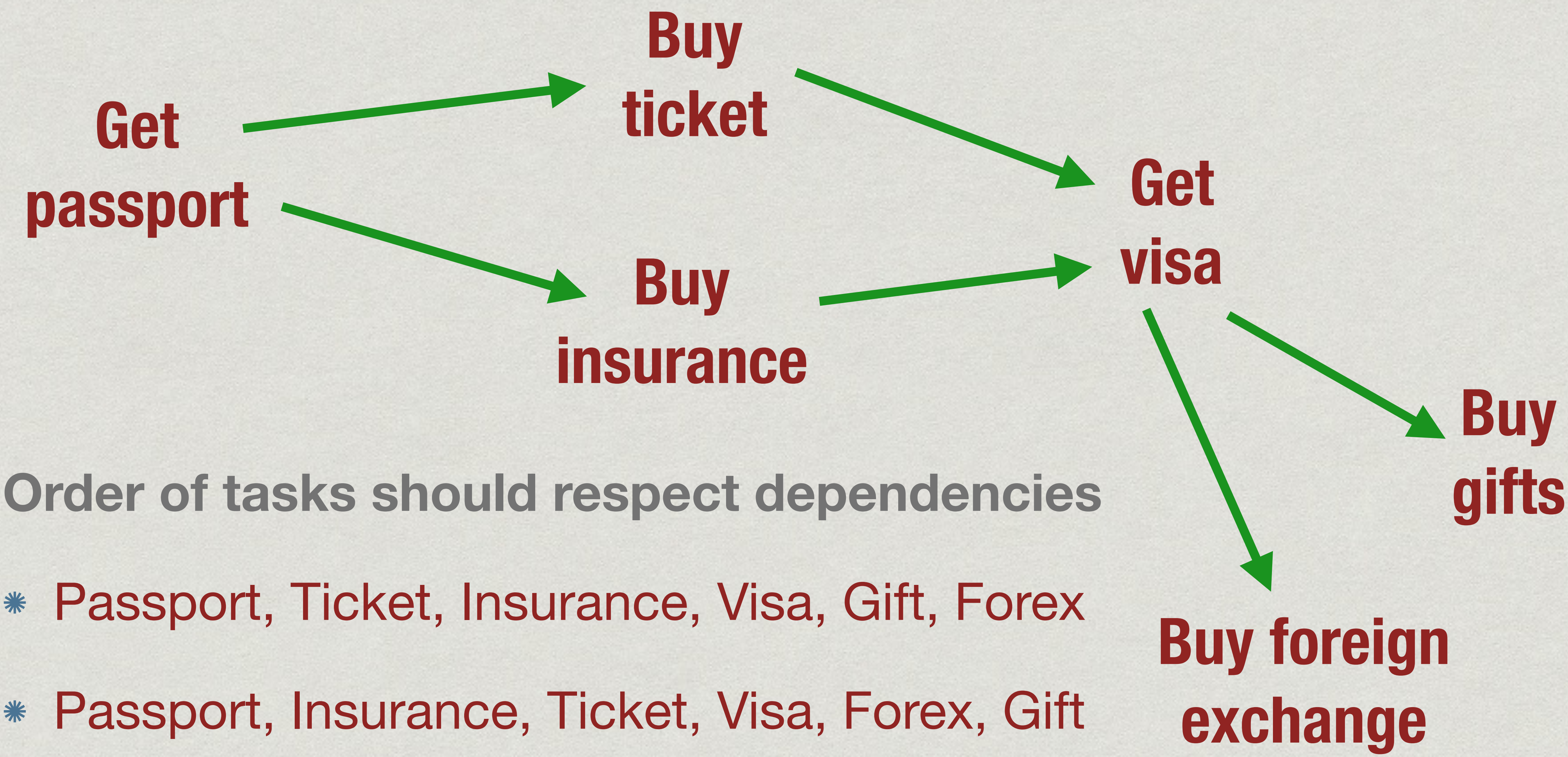
Our example as a graph



Order of tasks should respect dependencies

- Passport, Ticket, Insurance, Visa, Gift, Forex
- Passport, Insurance, Ticket, Visa, Forex, Gift
- Passport, Ticket, Insurance, Visa, Forex, Gift

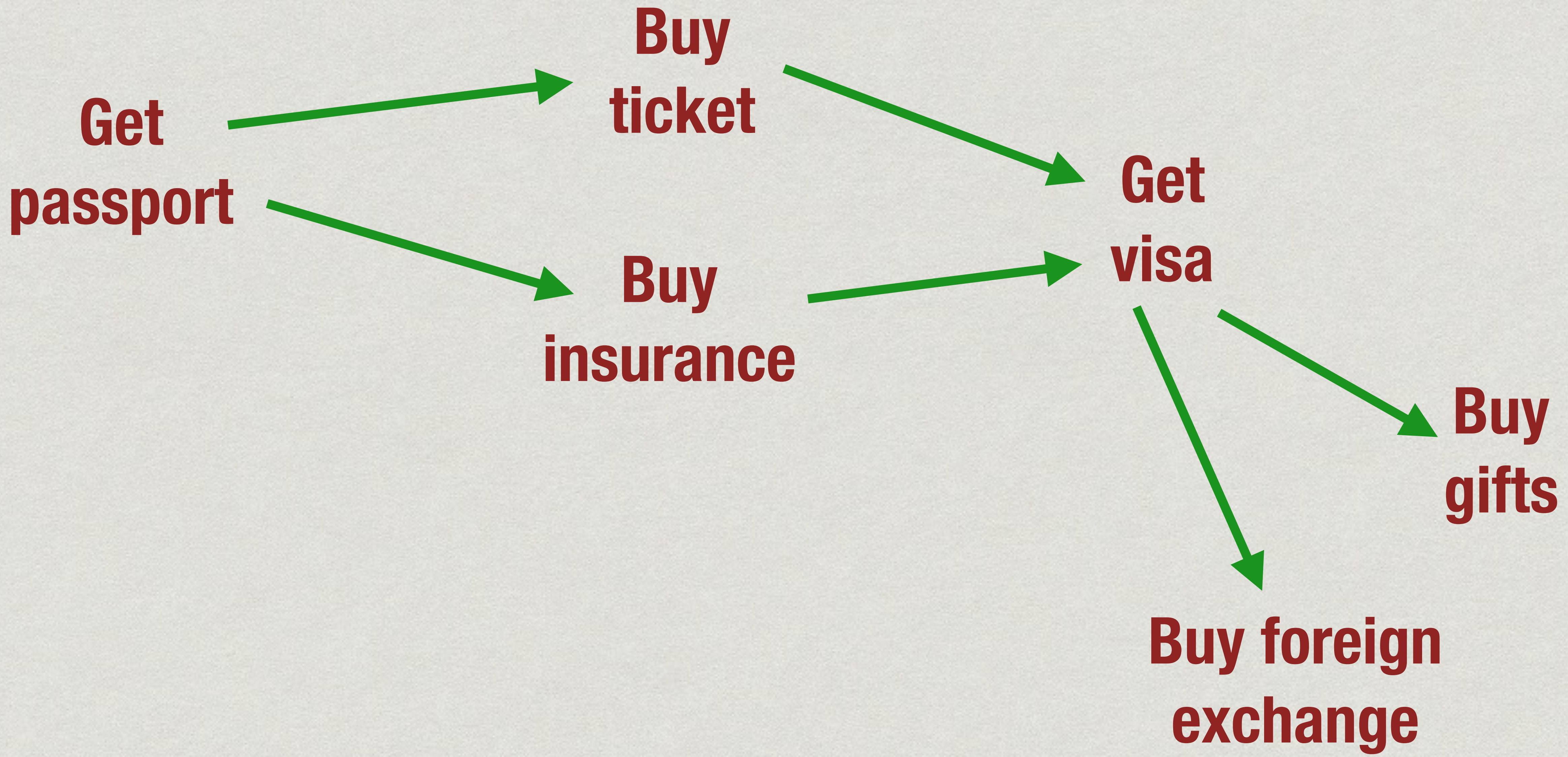
Our example as a graph



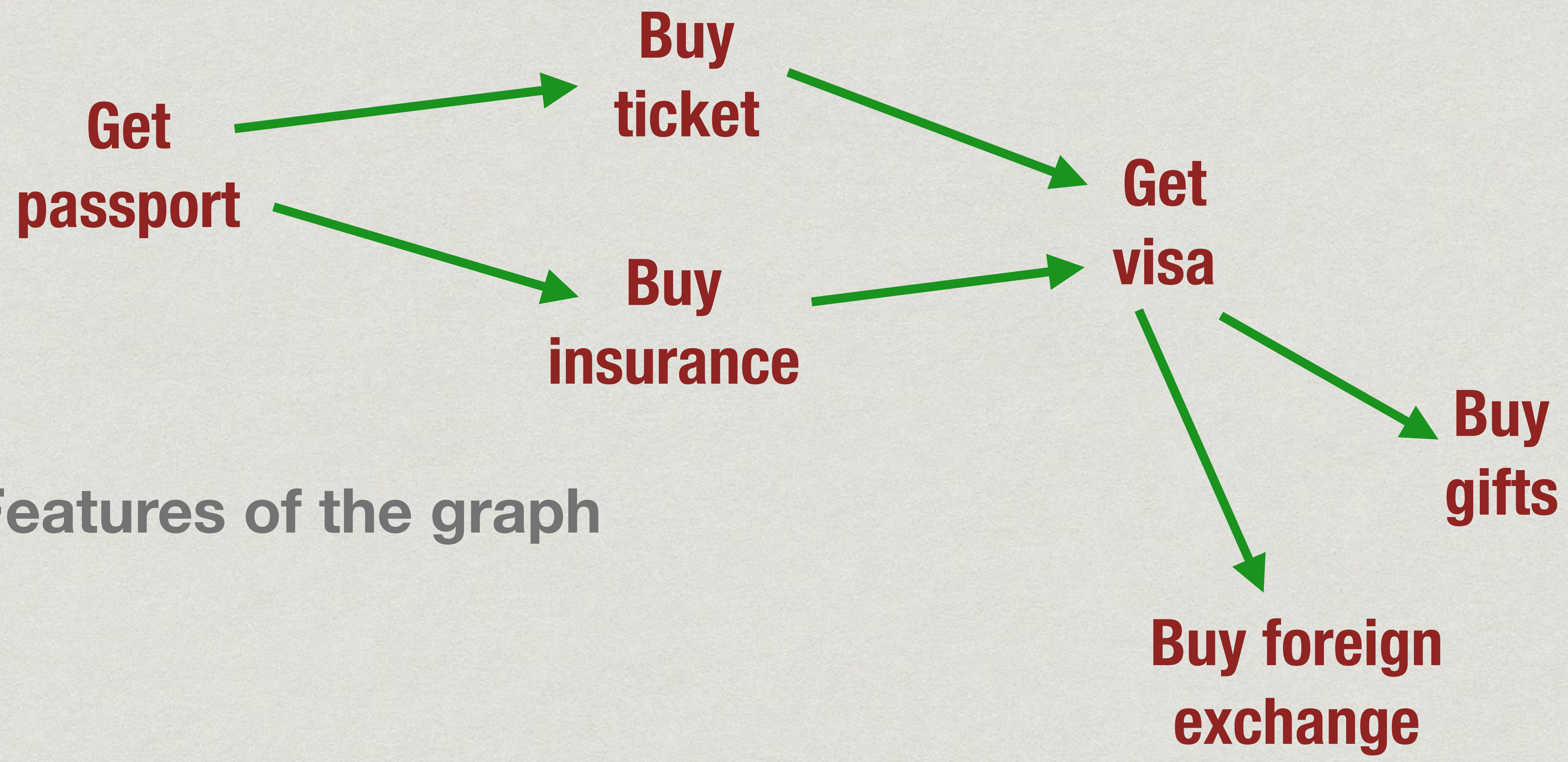
Order of tasks should respect dependencies

- Passport, Ticket, Insurance, Visa, Gift, Forex
- Passport, Insurance, Ticket, Visa, Forex, Gift
- Passport, Ticket, Insurance, Visa, Forex, Gift
- Passport, Insurance, Ticket, Visa, Gift, Forex

Our example as a graph

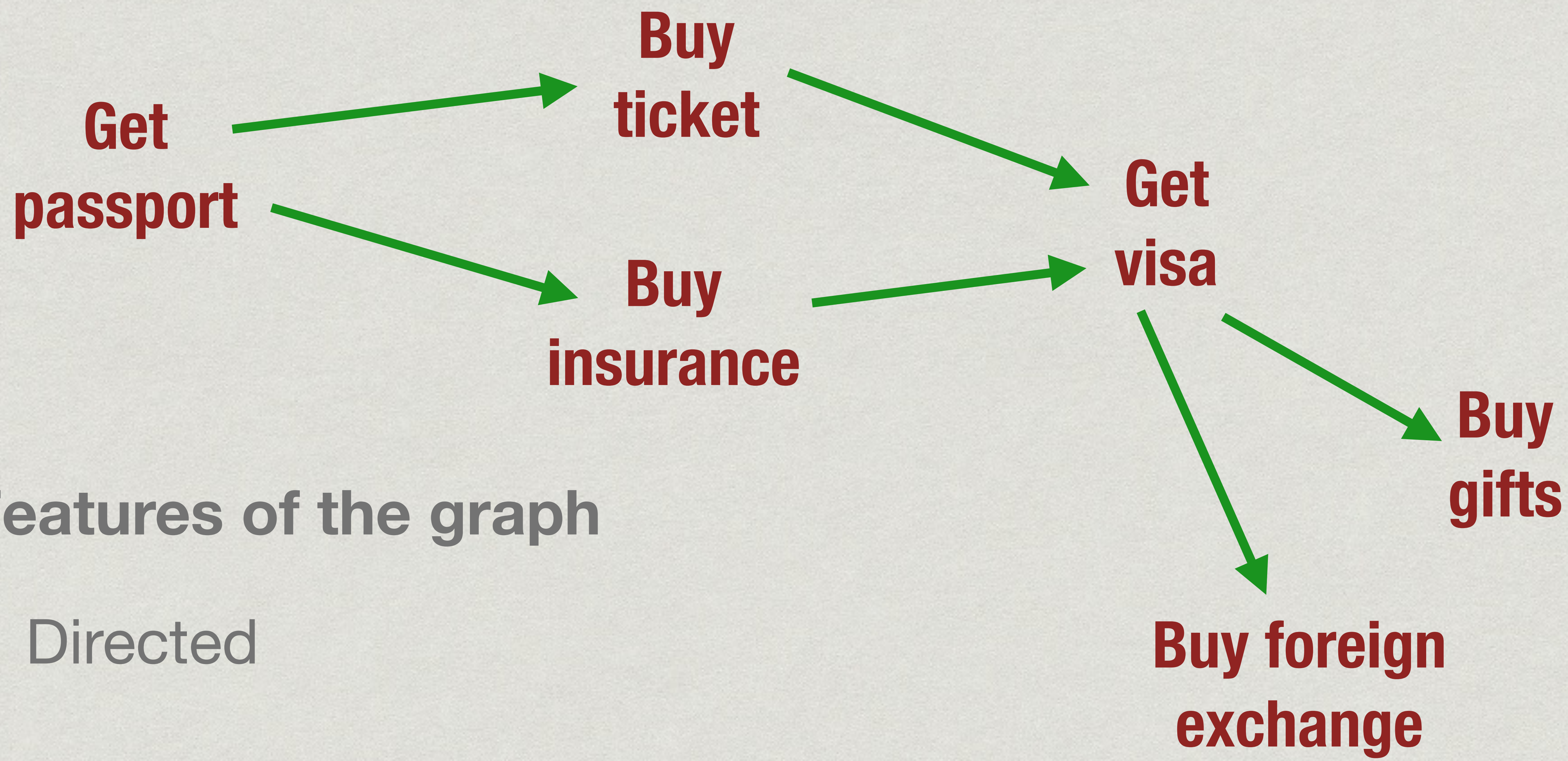


Our example as a graph



Features of the graph

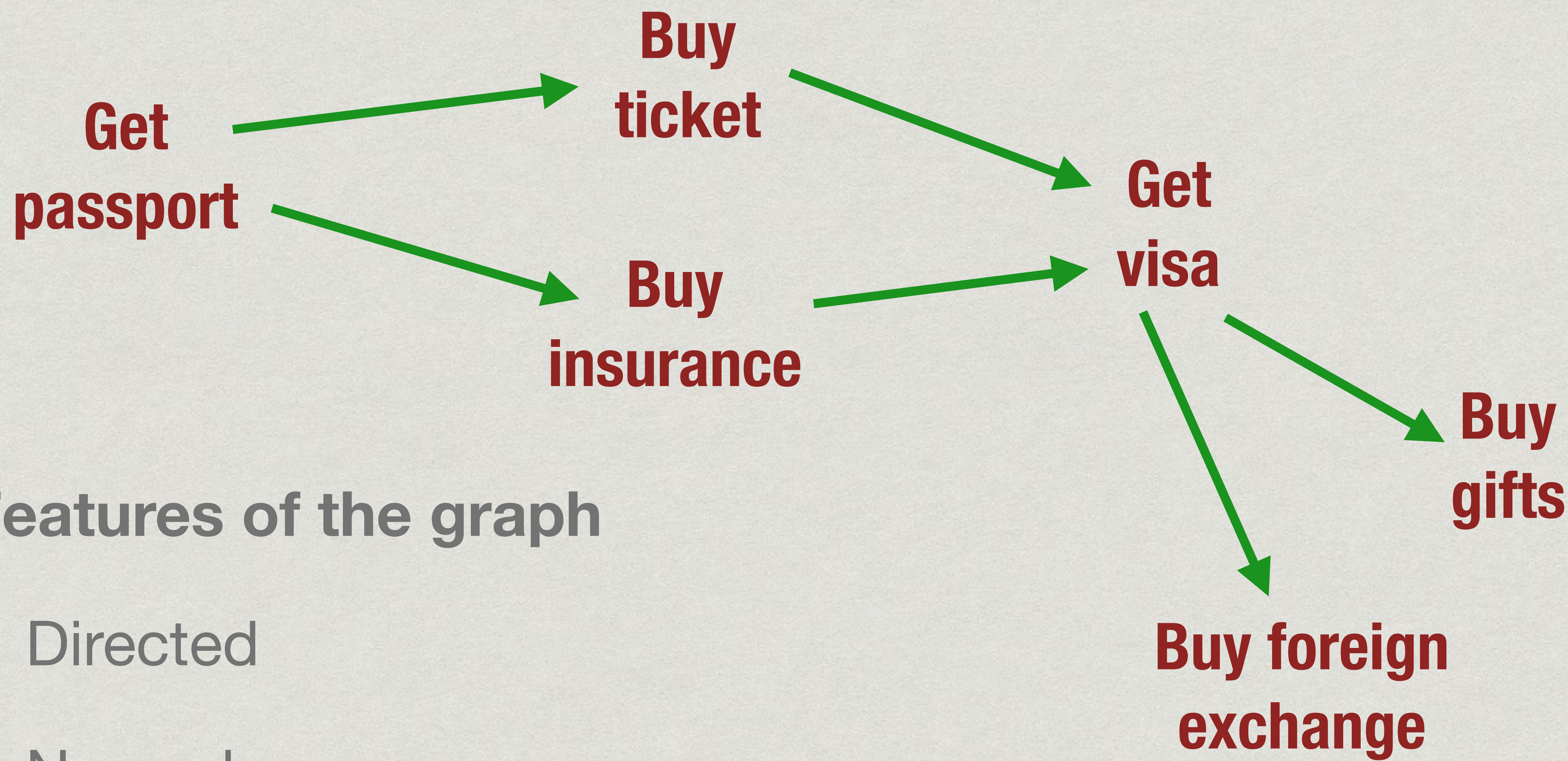
Our example as a graph



Features of the graph

- Directed

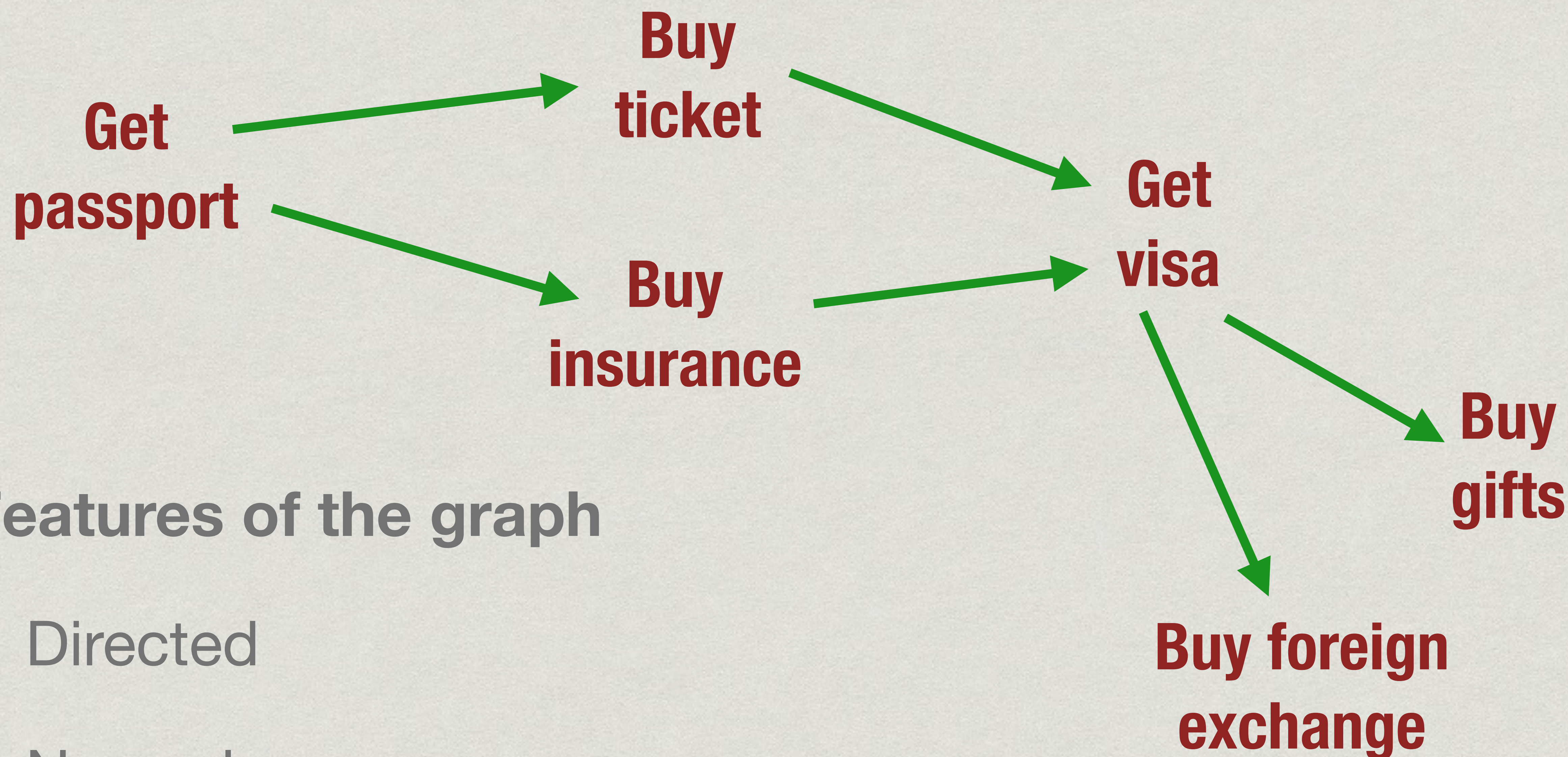
Our example as a graph



Features of the graph

- Directed
- No cycles

Our example as a graph



Features of the graph

- Directed
- No cycles
  - Cyclic dependencies are unsatisfiable

Directed Acyclic Graphs

Directed Acyclic Graphs

- $G = (V,E)$, a directed graph

Directed Acyclic Graphs

- G = (V,E), a directed graph
- No cycles

Directed Acyclic Graphs

- G = (V,E), a directed graph
- No cycles
  - No directed path from any v in V back to itself

Directed Acyclic Graphs

- G = (V,E), a directed graph
- No cycles
  - No directed path from any v in V back to itself
- Such graphs are also called DAGs

Topological ordering

Topological ordering

- Given a DAG $G = (V,E)$, $V = \{1,2,\ldots,n\}$
- Enumerate the vertices as $\{i_1,i_2,\ldots,i_n\}$ so that
  - For any edge (j,k) in E,

    j appears before k in the enumeration

Topological ordering

- Given a DAG $G = (V,E)$, $V = \{1,2,\ldots,n\}$
- Enumerate the vertices as $\{i_1,i_2,\ldots,i_n\}$ so that
  - For any edge (j,k) in E,

    j appears before k in the enumeration
- Also known as **topological sorting**

Topological ordering

Topological ordering

- **Observation**

Topological ordering

- **Observation**
  - A directed graph with cycles cannot be topologically ordered

Topological ordering

- **Observation**
  - A directed graph with cycles cannot be topologically ordered
  - Path from j to k and from k to j means
    - j must come before k
    - k must come before j
    - Impossible!

Topological ordering

Topological ordering

- **Claim**
  - Every directed acyclic graph can be topologically ordered

Topological ordering

- **Claim**
  - Every directed acyclic graph can be topologically ordered
- **Strategy**
  - First list vertices with no incoming edges
  - Then list vertices whose incoming neighbours are already listed
  - …

Topological ordering

Topological ordering

- indegree(v) : number of edges into v

Topological ordering

* indegree(v) : number of edges into v
* outdegree(v): number of edges out of v

Topological ordering

- indegree(v) : number of edges into v
- outdegree(v): number of edges out of v
- Every dag has at least one vertex with indegree 0

Topological ordering

- indegree(v) : number of edges into v
- outdegree(v): number of edges out of v
- Every dag has at least one vertex with indegree 0
  - Start with any v such that indegree(v) > 0

Topological ordering

- indegree(v) : number of edges into v
- outdegree(v): number of edges out of v
- Every dag has at least one vertex with indegree 0
  - Start with any v such that indegree(v) > 0
  - Walk backwards to a predecessor so long as indegree > 0

Topological ordering

- indegree(v) : number of edges into v
- outdegree(v): number of edges out of v
- Every dag has at least one vertex with indegree 0
  - Start with any v such that indegree(v) > 0
  - Walk backwards to a predecessor so long as indegree > 0
  - If no vertex has indegree 0, within n steps we will complete a cycle!

Topological ordering

Topological ordering

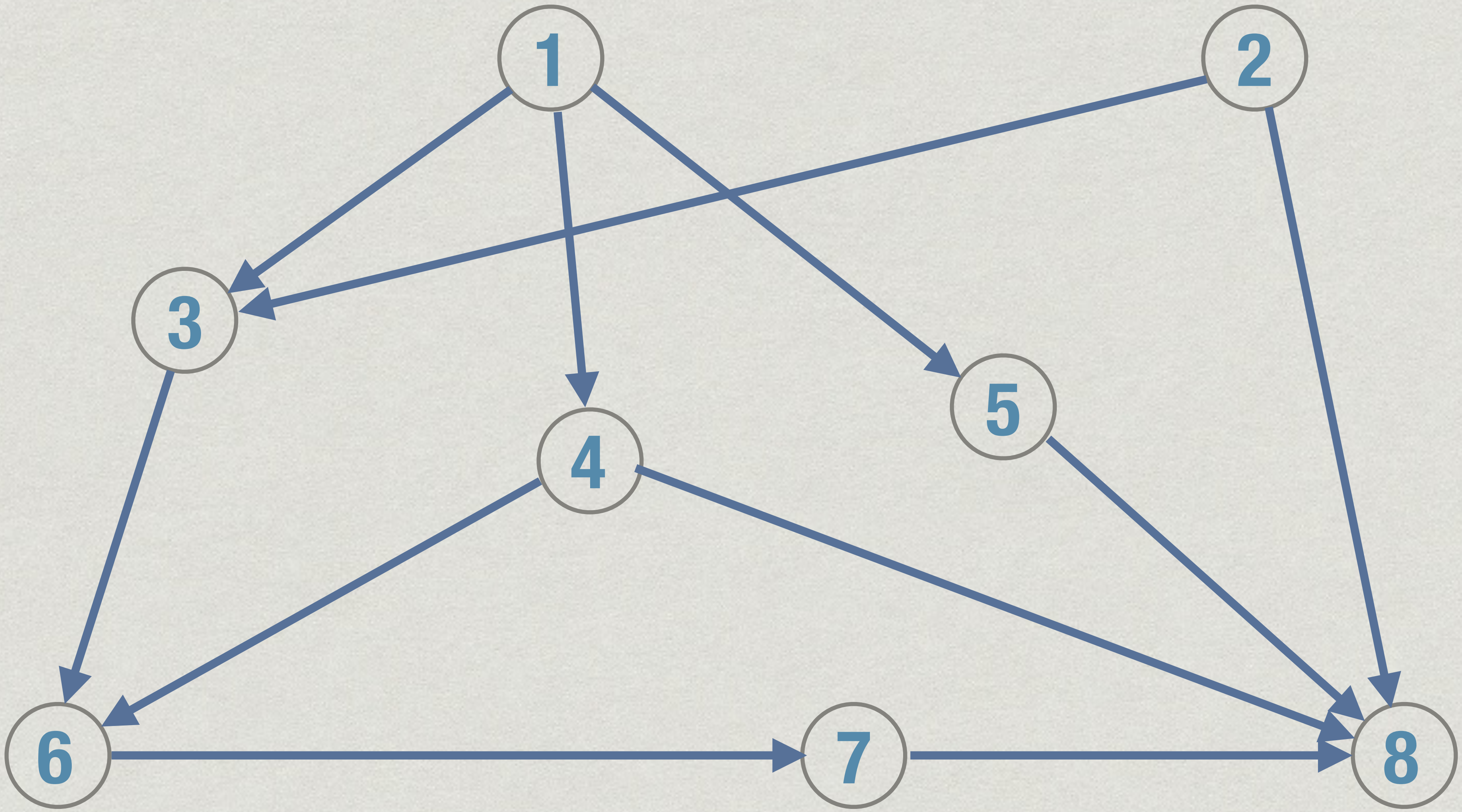
- Pick a vertex with indegree 0
  - No dependencies
  - Enumerate it and delete it from the graph

Topological ordering

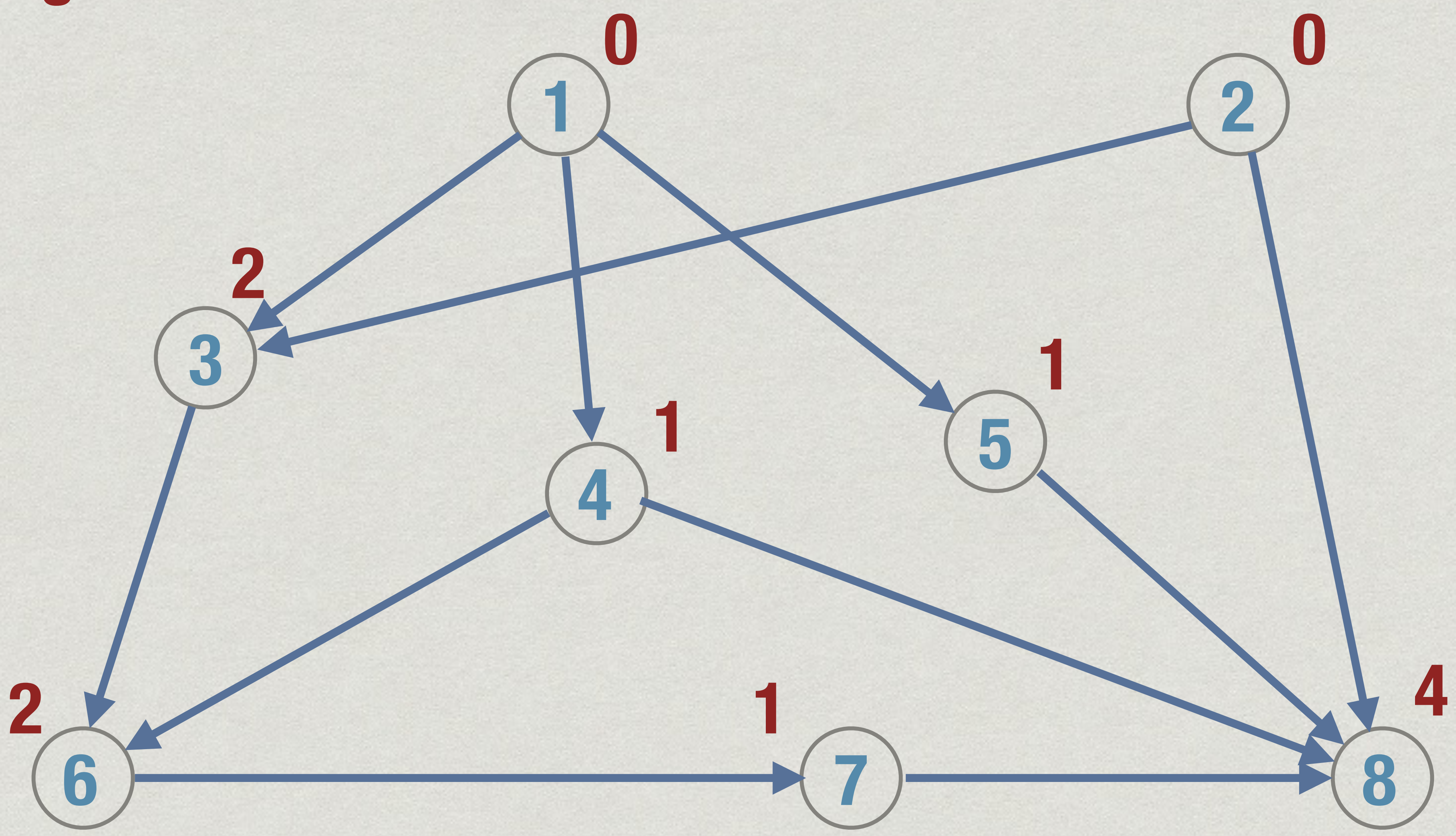
- Pick a vertex with indegree 0
  - No dependencies
  - Enumerate it and delete it from the graph
- What remains is again a DAG!

Topological ordering

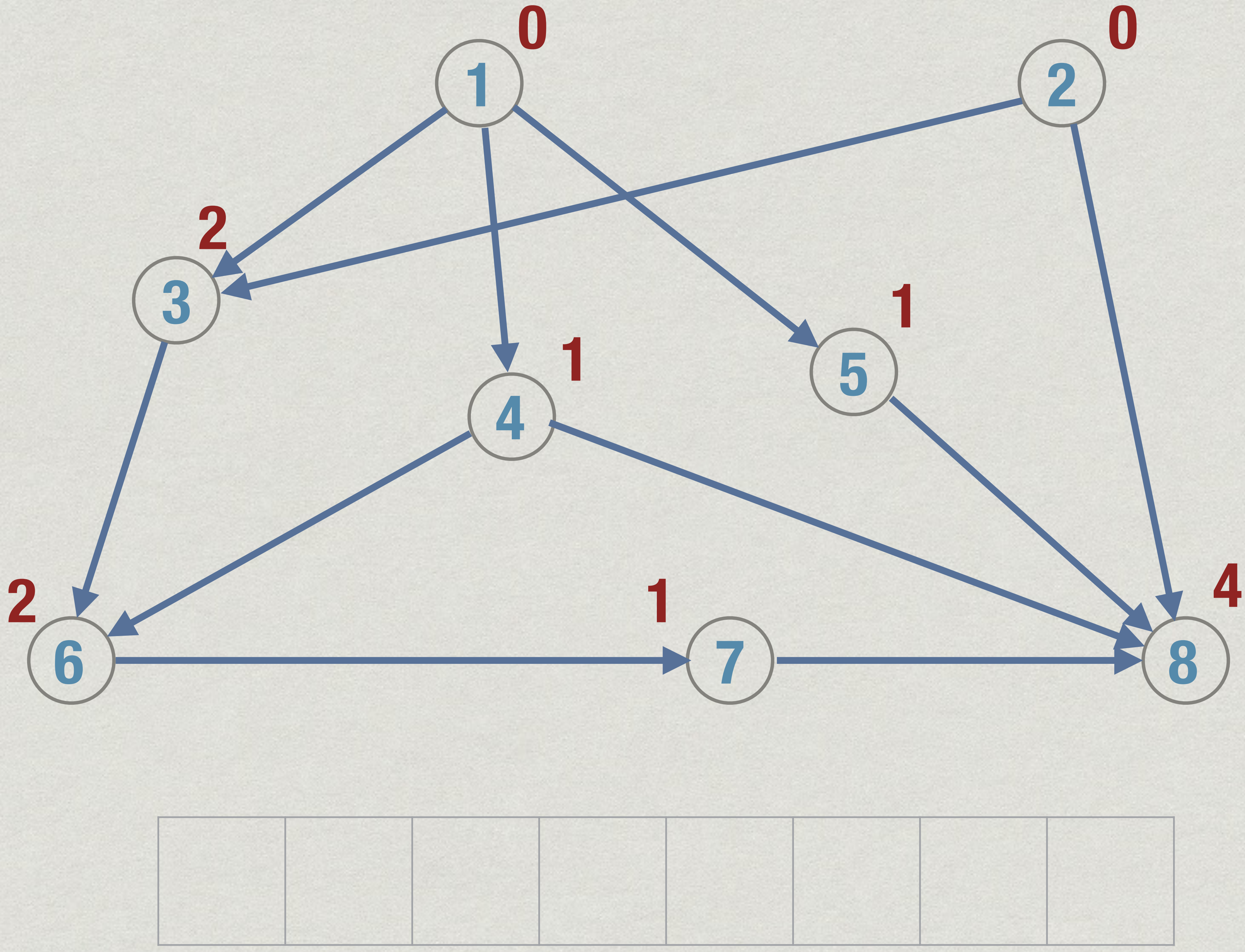
- Pick a vertex with indegree 0
  - No dependencies
  - Enumerate it and delete it from the graph
- What remains is again a DAG!
- Repeat the step above
  - Stop when the resulting DAG is empty


1
2
3
4
5
6
7
8

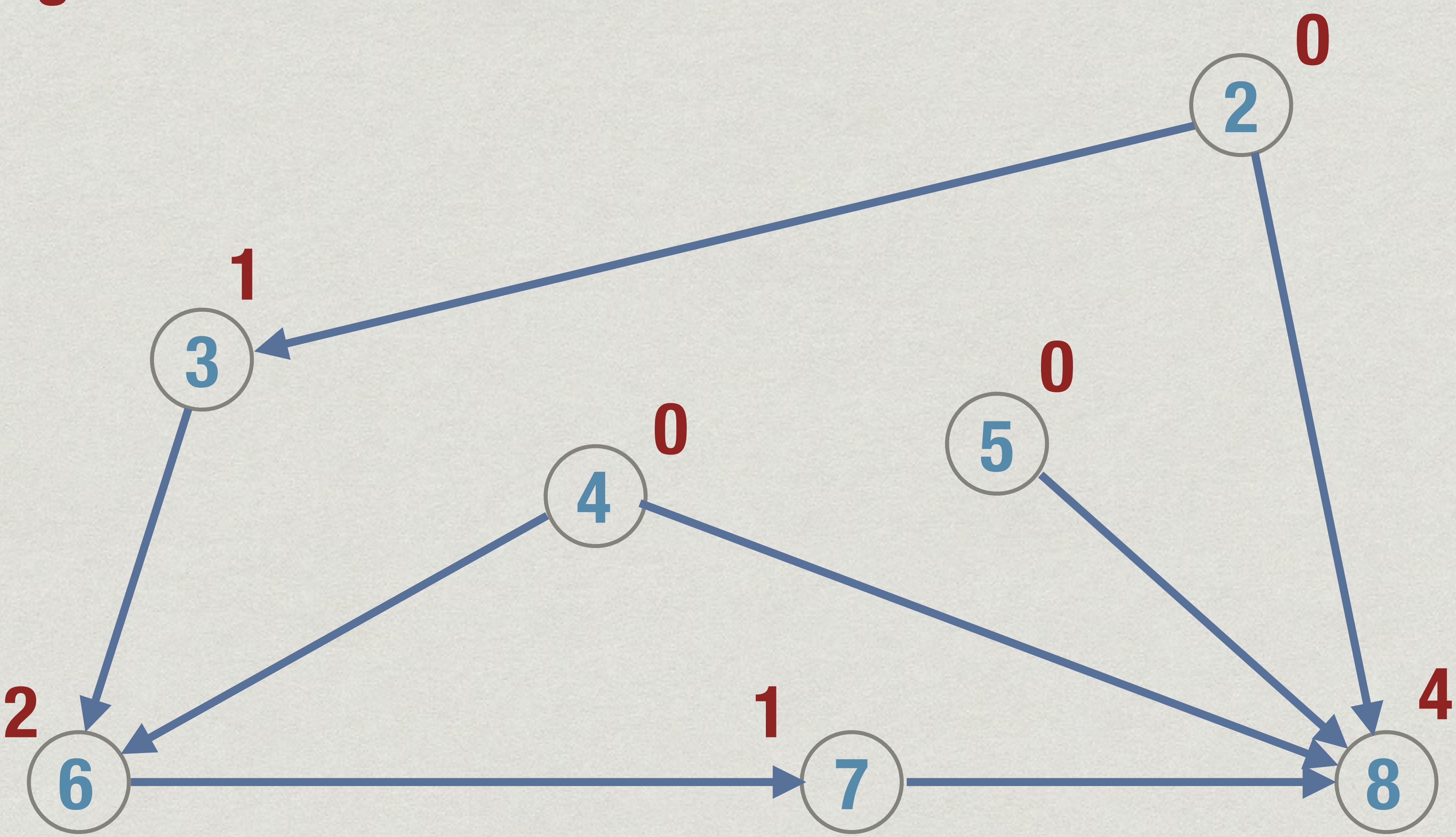
Indegree



Indegree

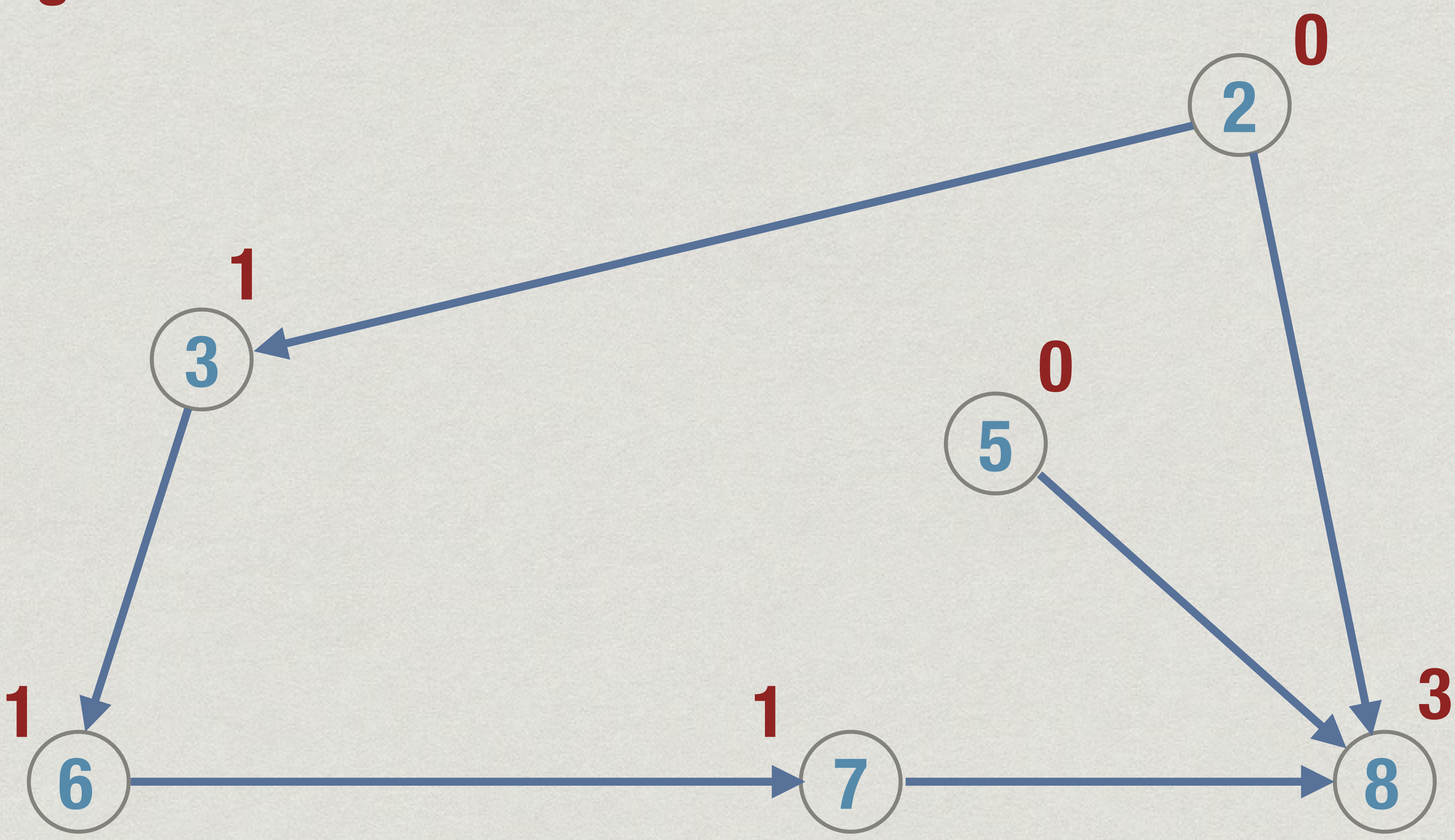


Indegree



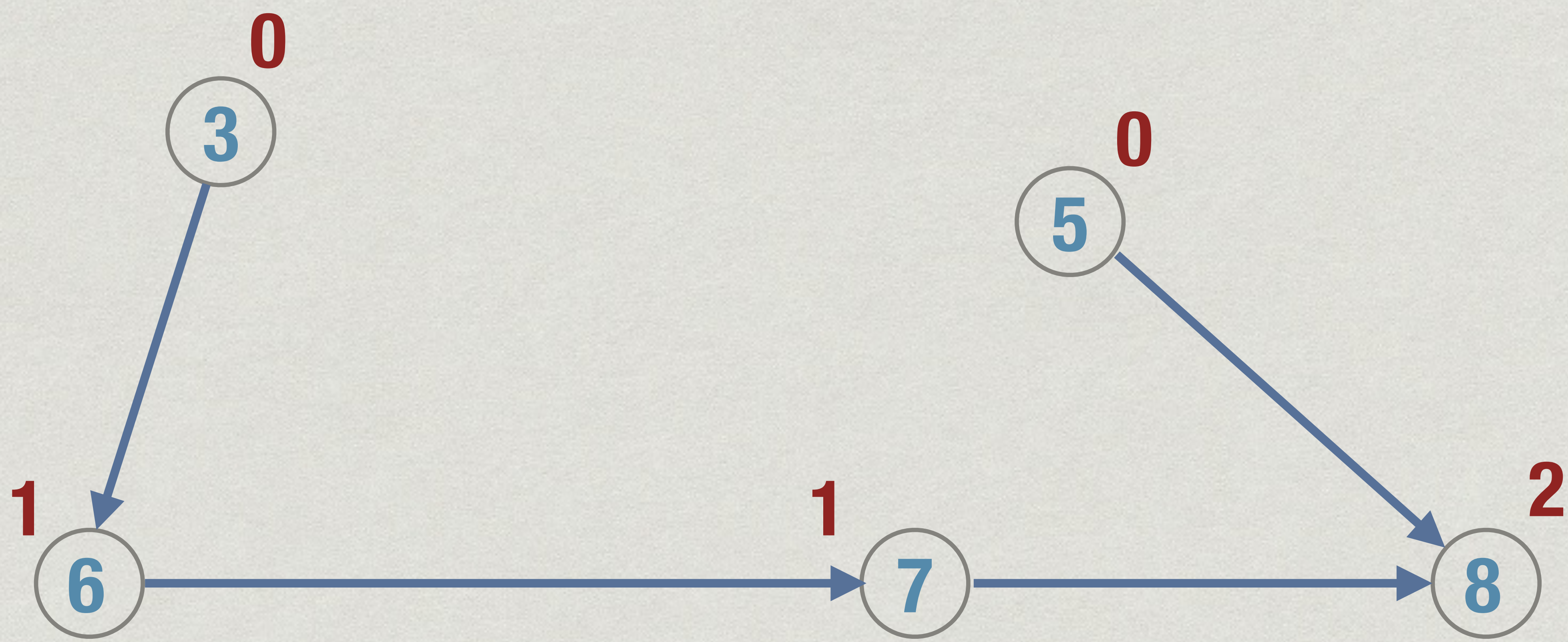
1							

Indegree



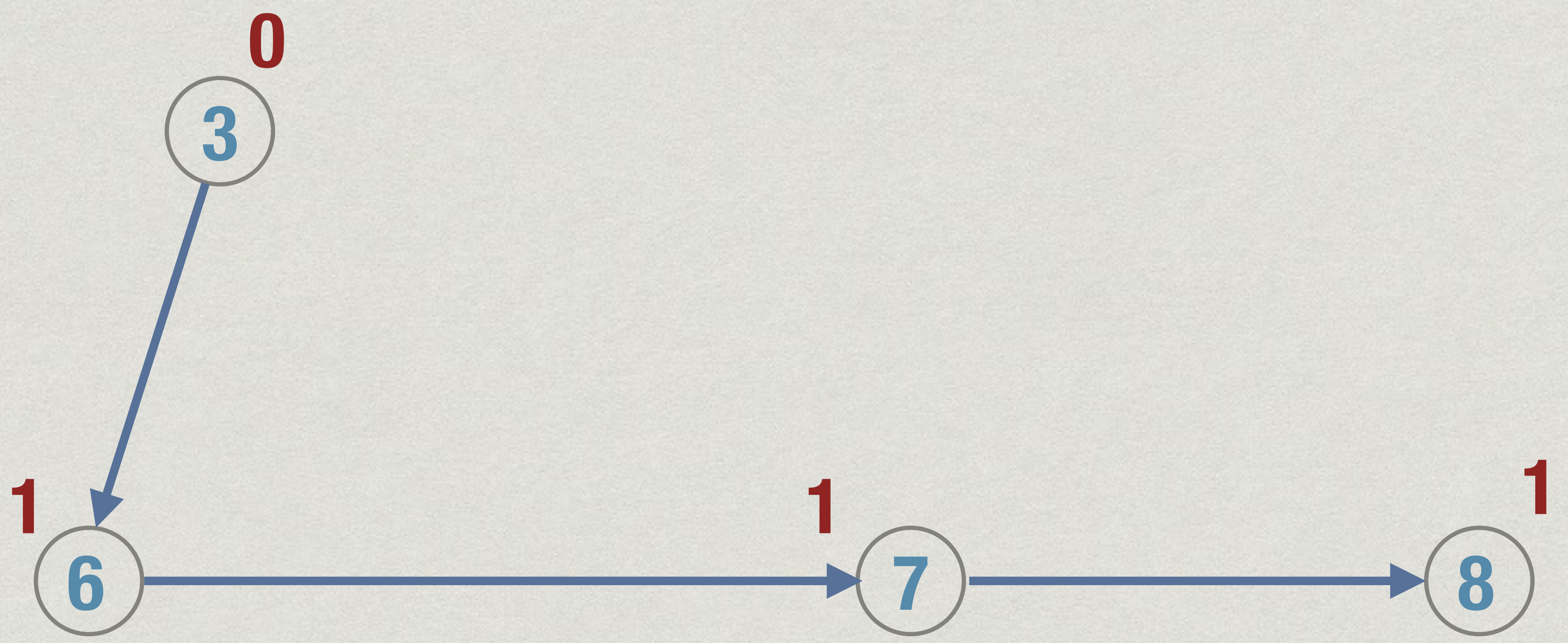
1	4						

Indegree



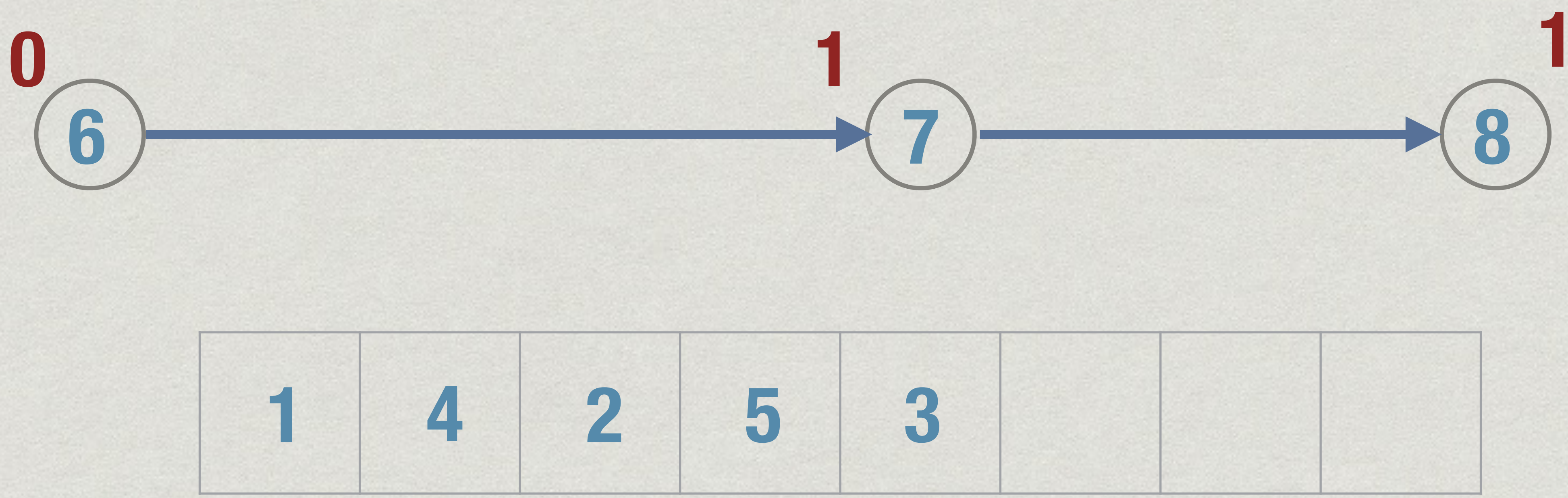
1	4	2					

Indegree

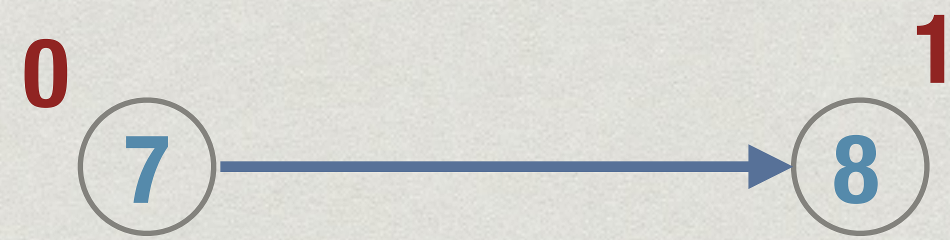


1	4	2	5				

Indegree



Indegree



1	4	2	5	3	6		

Indegree

0

8

1	4	2	5	3	6	7	

Indegree

1	4	2	5	3	6	7	8

Topological ordering

```
function TopologicalOrder(G)
  for i = 1 to n
    indegree[i] = 0
    for j = 1 to n
      indegree[i] = indegree[i] + A[j][i]

  for i = 1 to n
    choose j with indegree[j] = 0
      enumerate j
      indegree[j] = -1
      for k = 1 to n
        if A[j][k] == 1
          indegree[k] = indegree[k]-1
```

Topological ordering

Topological ordering

- Complexity is $O(n^2)$

Topological ordering

- Complexity is $O(n^2)$
  - Initializing indegree takes time $O(n^2)$

Topological ordering

- Complexity is $O(n^2)$
  - Initializing indegree takes time $O(n^2)$
  - Loop n times to enumerate vertices
    - Inside loop, identifying next vertex is $O(n)$
    - Updating indegrees of neighbours is $O(n)$

Topological ordering

Topological ordering

* Using adjacency list

Topological ordering

- Using adjacency list
  - Scan lists once to compute indegrees — O(m)

Topological ordering

- Using adjacency list
  - Scan lists once to compute indegrees — $O(m)$
  - Put all indegree 0 vertices in a queue

Topological ordering

- Using adjacency list
  - Scan lists once to compute indegrees — O(m)
  - Put all indegree 0 vertices in a queue
  - Enumerate head of queue and decrement indegree of neighbours — degree(j), overall O(m)

Topological ordering

- Using adjacency list
  - Scan lists once to compute indegrees — O(m)
  - Put all indegree 0 vertices in a queue
  - Enumerate head of queue and decrement indegree of neighbours — degree(j), overall O(m)
    - If indegree(k) becomes 0, add to queue

Topological ordering

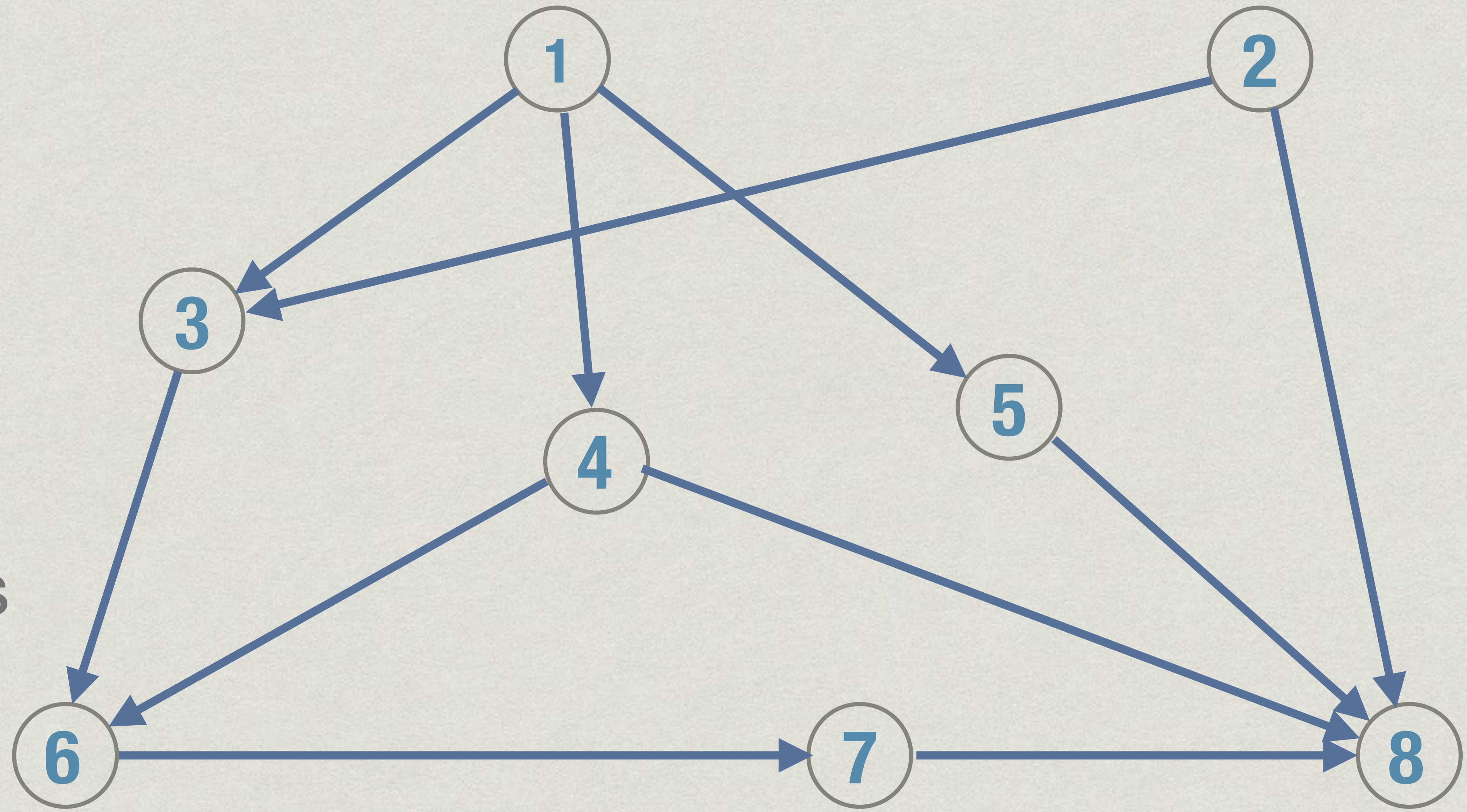
- Using adjacency list
  - Scan lists once to compute indegrees — $O(m)$
  - Put all indegree 0 vertices in a queue
  - Enumerate head of queue and decrement indegree of neighbours — degree(j), overall $O(m)$
    - If indegree(k) becomes 0, add to queue
- Overall $O(n+m)$

Topological ordering revisited

```
function TopologicalOrder(G) //Edges are in adjacency list
  for i = 1 to n { indegree[i] = 0 }

  for i = 1 to n
    for (i,j) in E //proportional to outdegree(i)
      indegree[j] = indegree[j] + 1

  for i = 1 to n
    if indegree[i] == 0 { add i to Queue }

  while Queue is not empty
    j = remove_head(Queue)
    for (j,k) in E //proportional to outdegree(j)
      indegree[k] = indegree[k] - 1
      if indegree[k] == 0 { add k to Queue }
```

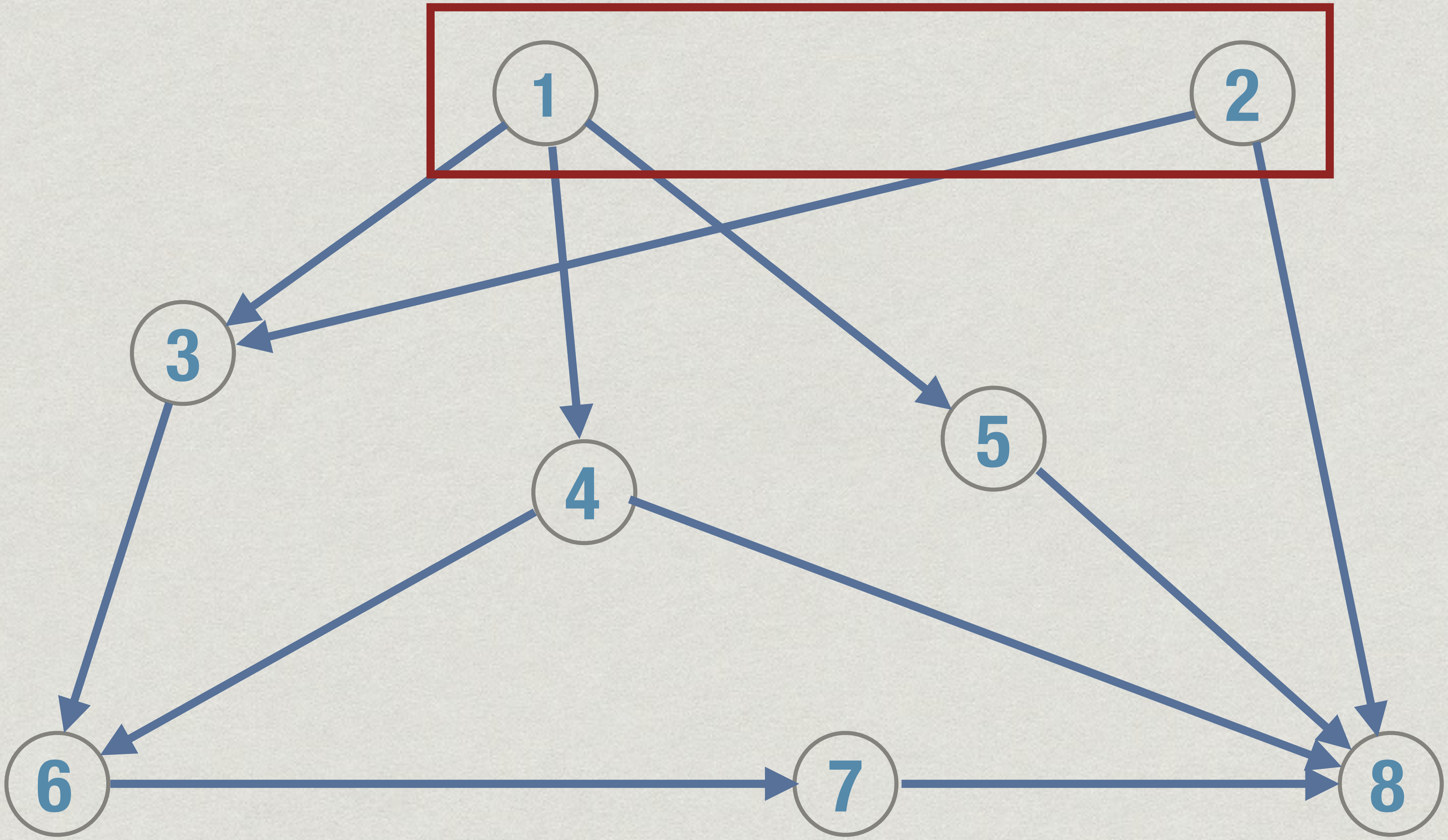
Topological ordering

- Imagine these are courses
- Edges are pre-requisites



- What is the minimum number of semesters to complete the programme?

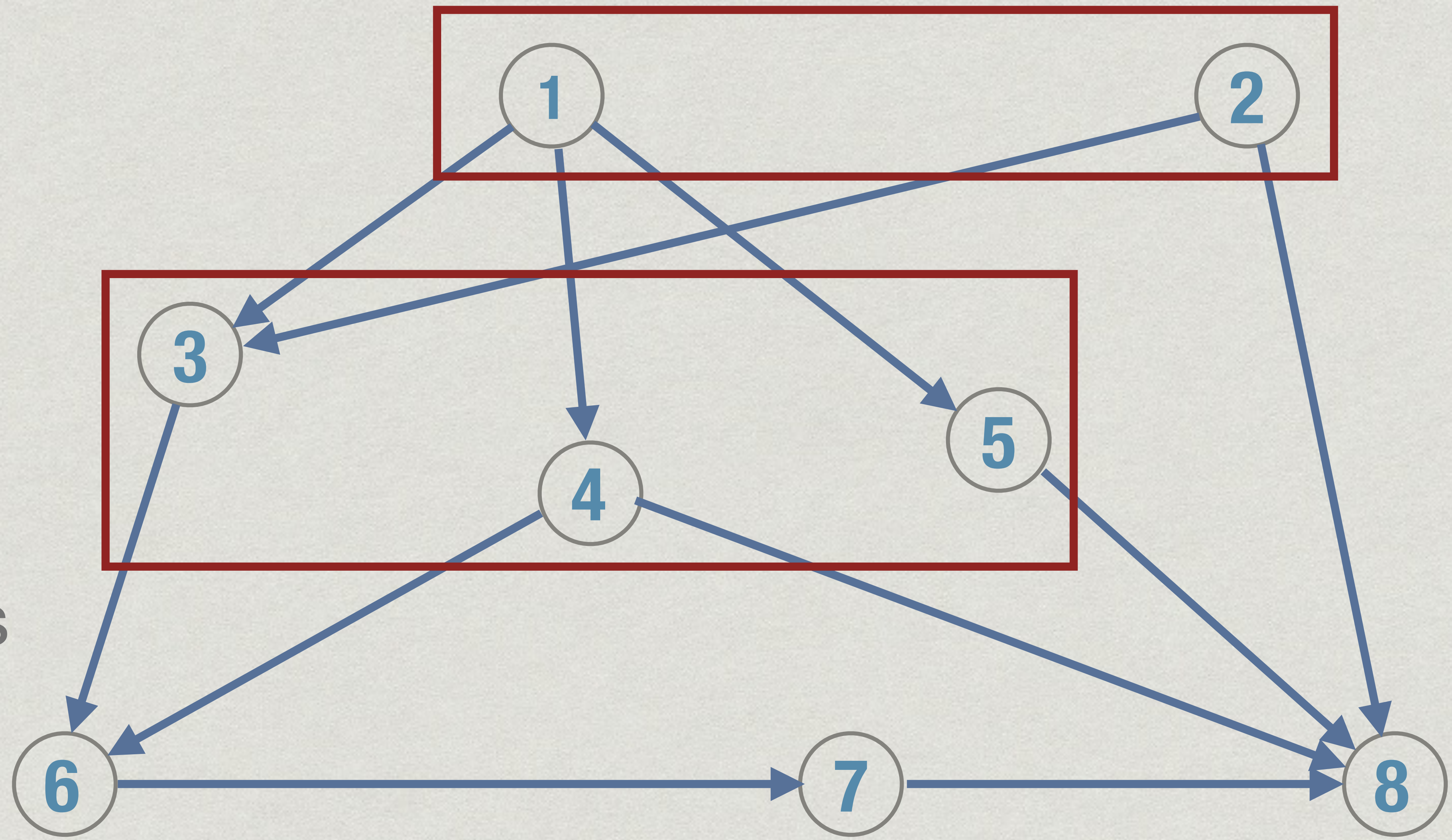
Topological ordering



- Imagine these are courses
- Edges are pre-requisites
- What is the minimum number of semesters to complete the programme?

Topological ordering

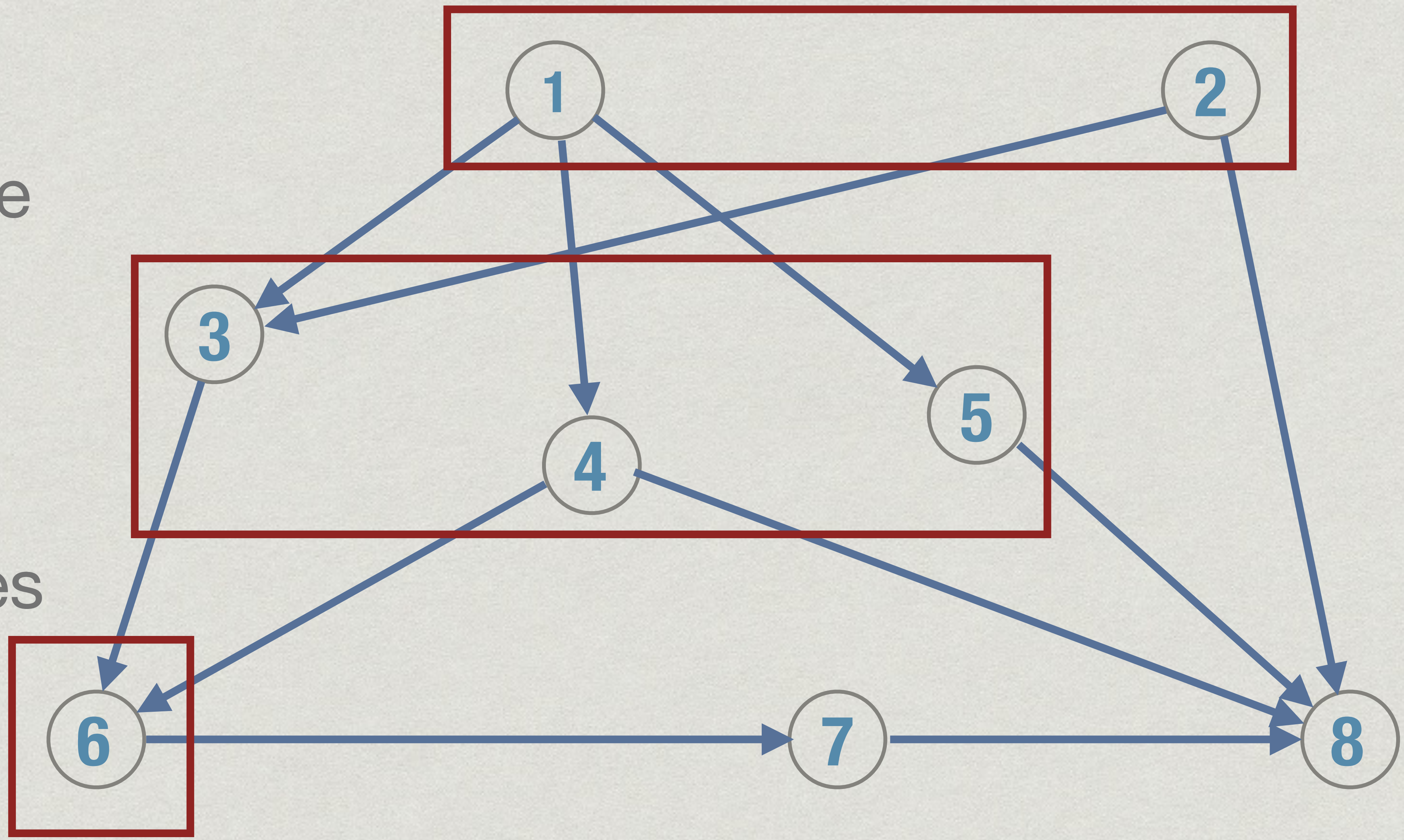
- Imagine these are courses
- Edges are pre-requisites



- What is the minimum number of semesters to complete the programme?

Topological ordering

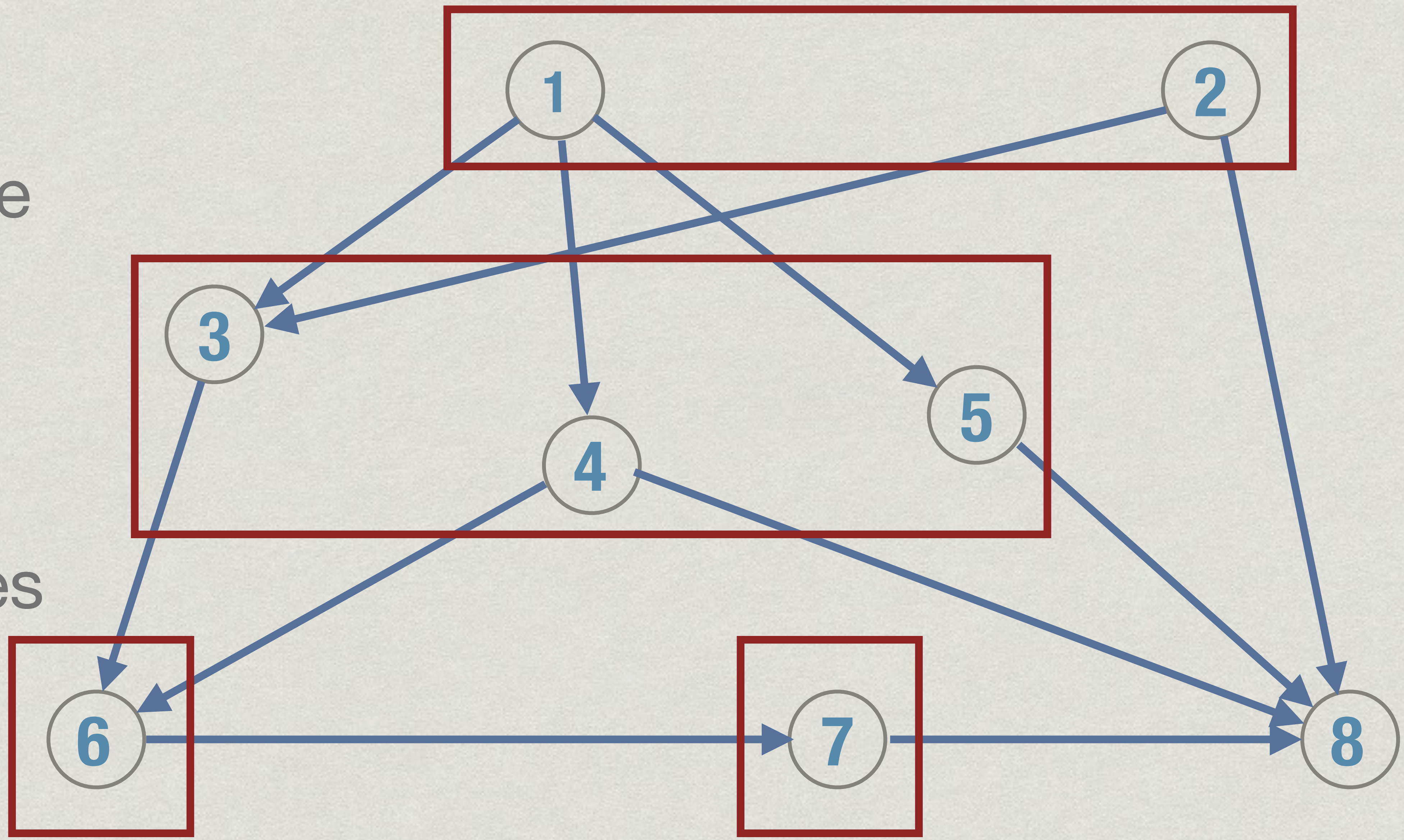
- Imagine these are courses
- Edges are pre-requisites



- What is the minimum number of semesters to complete the programme?

Topological ordering

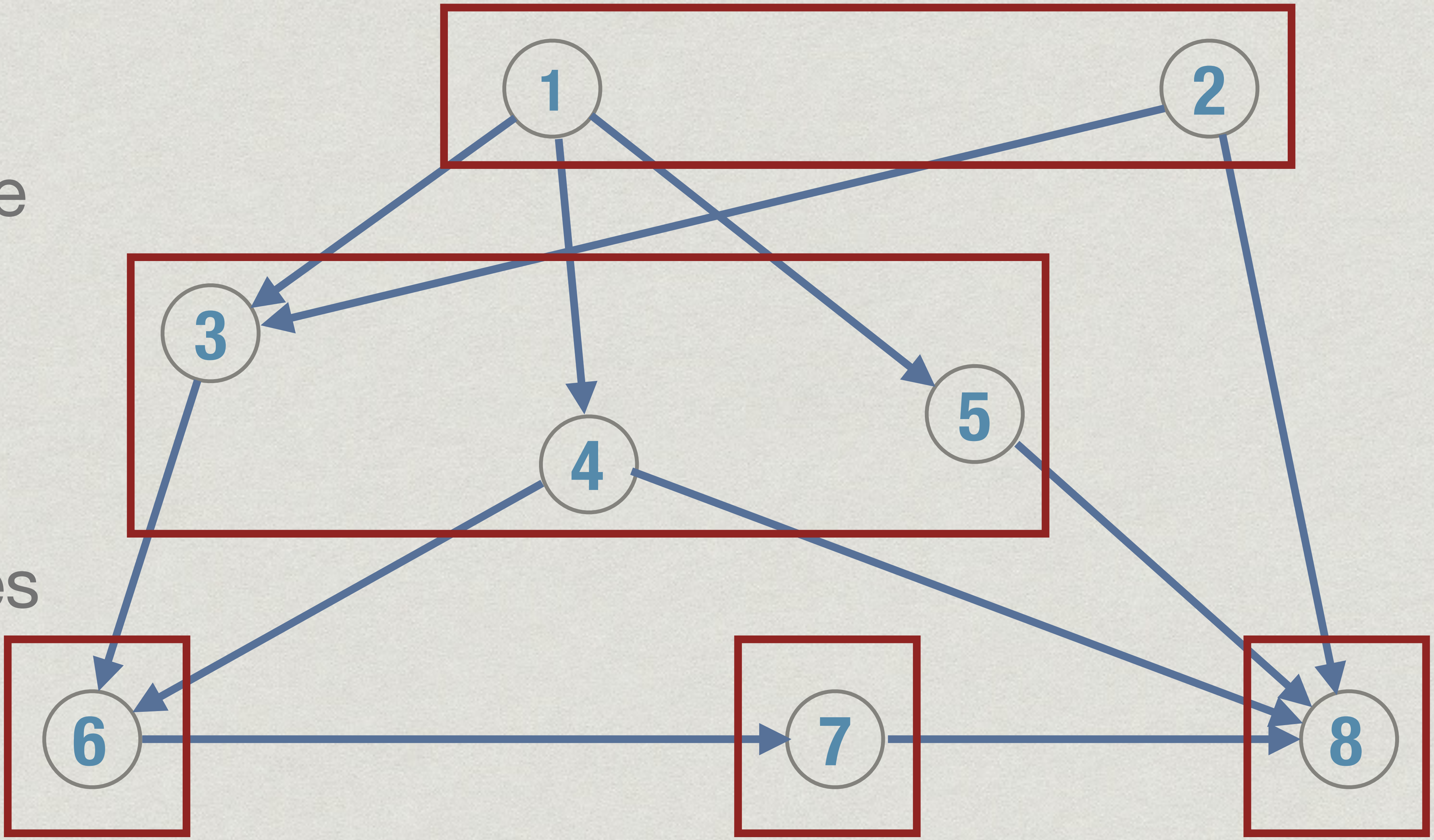
- Imagine these are courses
- Edges are pre-requisites



- What is the minimum number of semesters to complete the programme?

Topological ordering

- Imagine these are courses
- Edges are pre-requisites



- What is the minimum number of semesters to complete the programme?

Longest path in a DAG

Longest path in a DAG

- Equivalent to finding longest path in the DAG

Longest path in a DAG

- Equivalent to finding longest path in the DAG
- If indegree(j) = 0, longest_path_to(j) = 0

Longest path in a DAG

- Equivalent to finding longest path in the DAG
- If indegree(j) = 0, longest_path_to(j) = 0
- If indegree(k) > 0, longest_path_to(k) is

  1 + max{ longest_path_to(j) } among all

  incoming neighbours j of k

Longest path in a DAG

Longest path in a DAG

- To compute longest_path_to(k)
  - Need longest_path_to(j) for all incoming neighbours of k

Longest path in a DAG

- To compute longest_path_to(k)
  - Need longest_path_to(j) for all incoming neighbours of k
- If j is an incoming neighbour, (j,k) in E
  - j is enumerated before k in topological order

Longest path in a DAG

- To compute longest_path_to(k)
  - Need longest_path_to(j) for all incoming neighbours of k
- If j is an incoming neighbour, (j,k) in E
  - j is enumerated before k in topological order
- Hence, compute longest_path_to(i) in topological order

Longest path in a DAG

Longest path in a DAG

- Let $i_1, i_2, \ldots, i_n$ be a topological ordering of V

Longest path in a DAG

- Let $i_1, i_2, \ldots, i_n$ be a topological ordering of V
- All neighbours of i_k appear before it in this list

Longest path in a DAG

- Let $i_1, i_2, \ldots, i_n$ be a topological ordering of V
- All neighbours of i_k appear before it in this list
- From left to right, compute longest_path_to(i_k) as

  1 + max{ longest_path_to(i_j) } among all

  incoming neighbours i_j of i_k

Longest path in a DAG

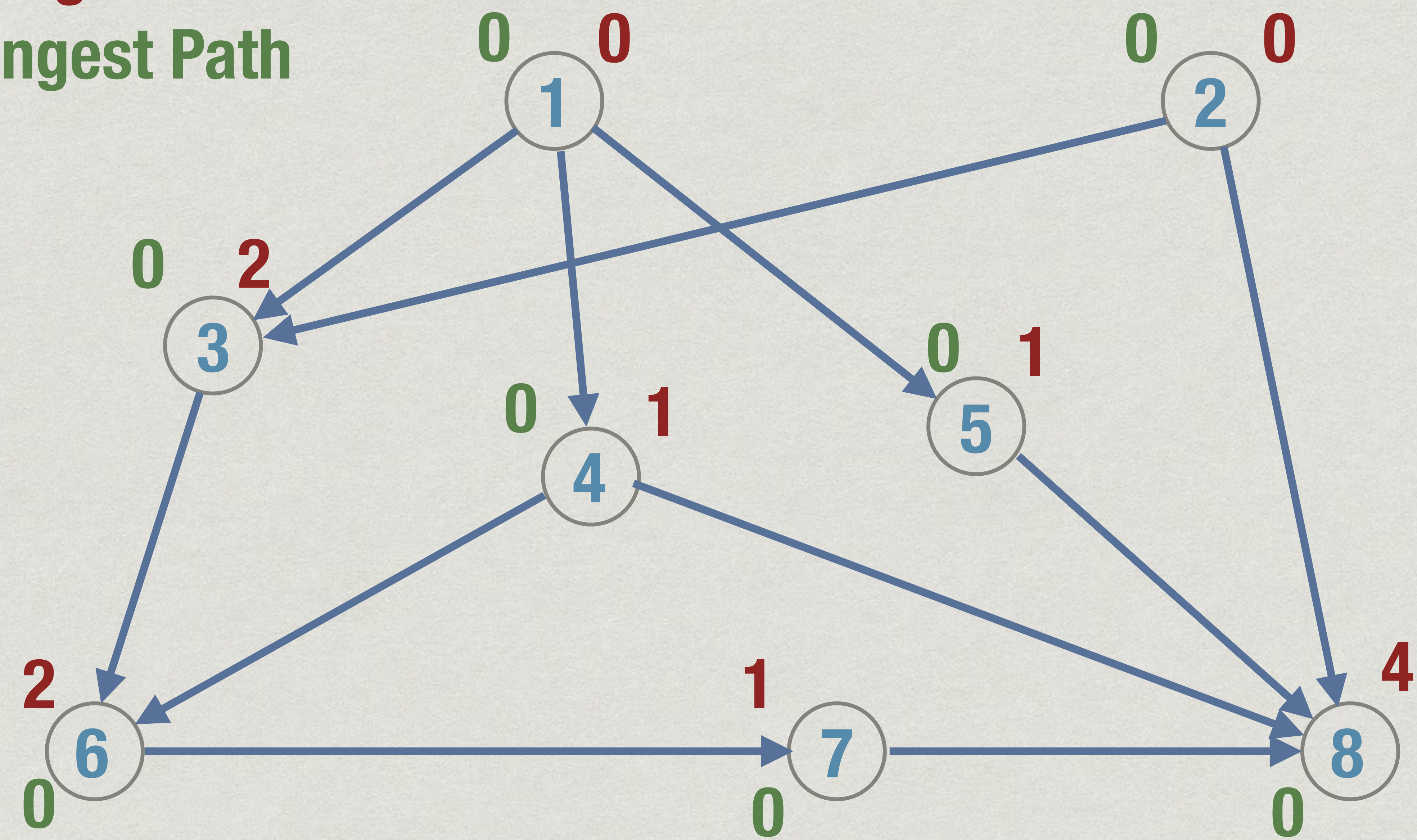
- Let $i_1, i_2, \ldots, i_n$ be a topological ordering of V
- All neighbours of i_k appear before it in this list
- From left to right, compute longest_path_to(i_k) as

    1 + max{ longest_path_to(i_j) } among all

    incoming neighbours i_j of i_k
- Can combine this calculation with topological sort

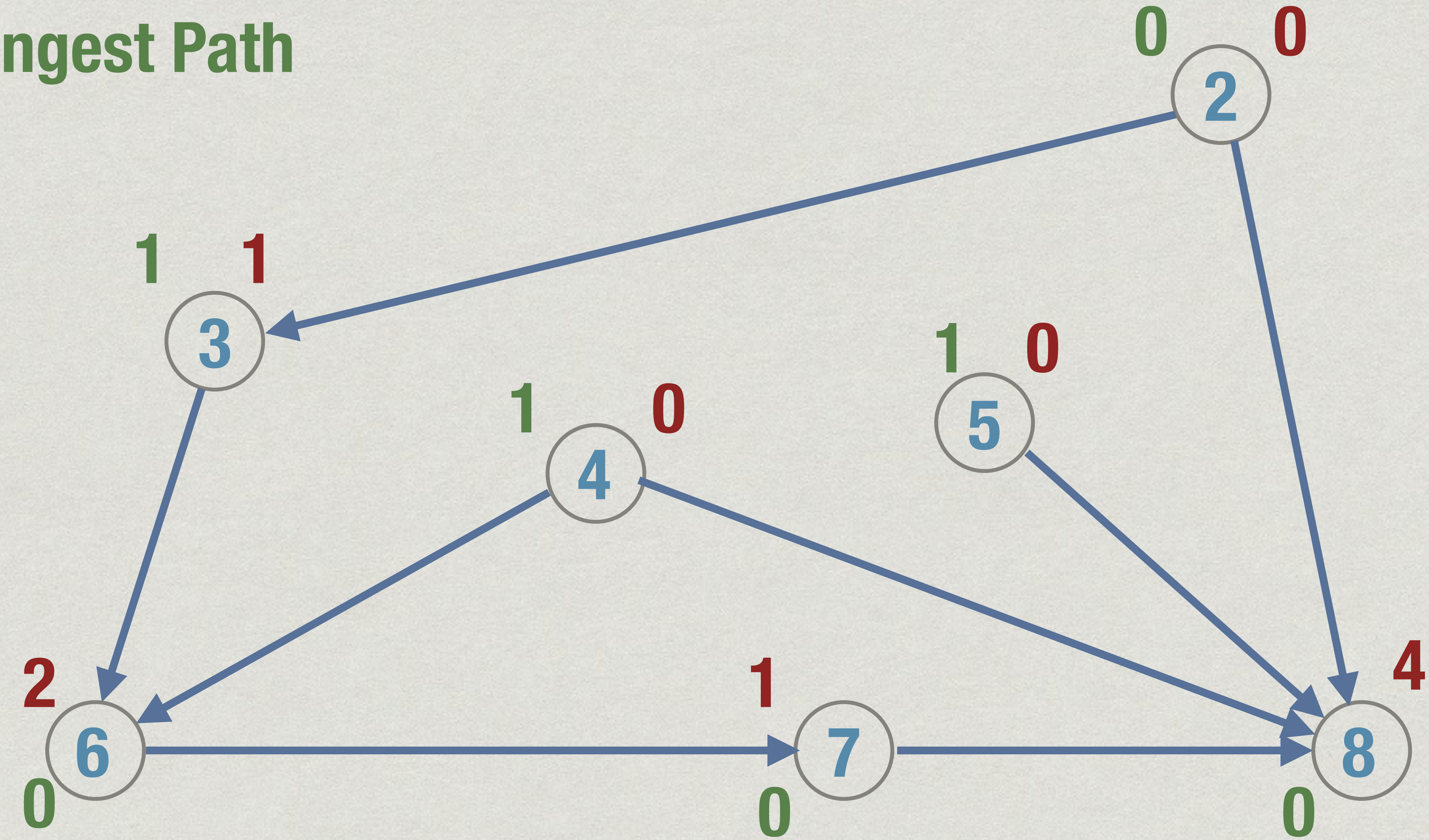
Indegree

Longest Path



Indegree

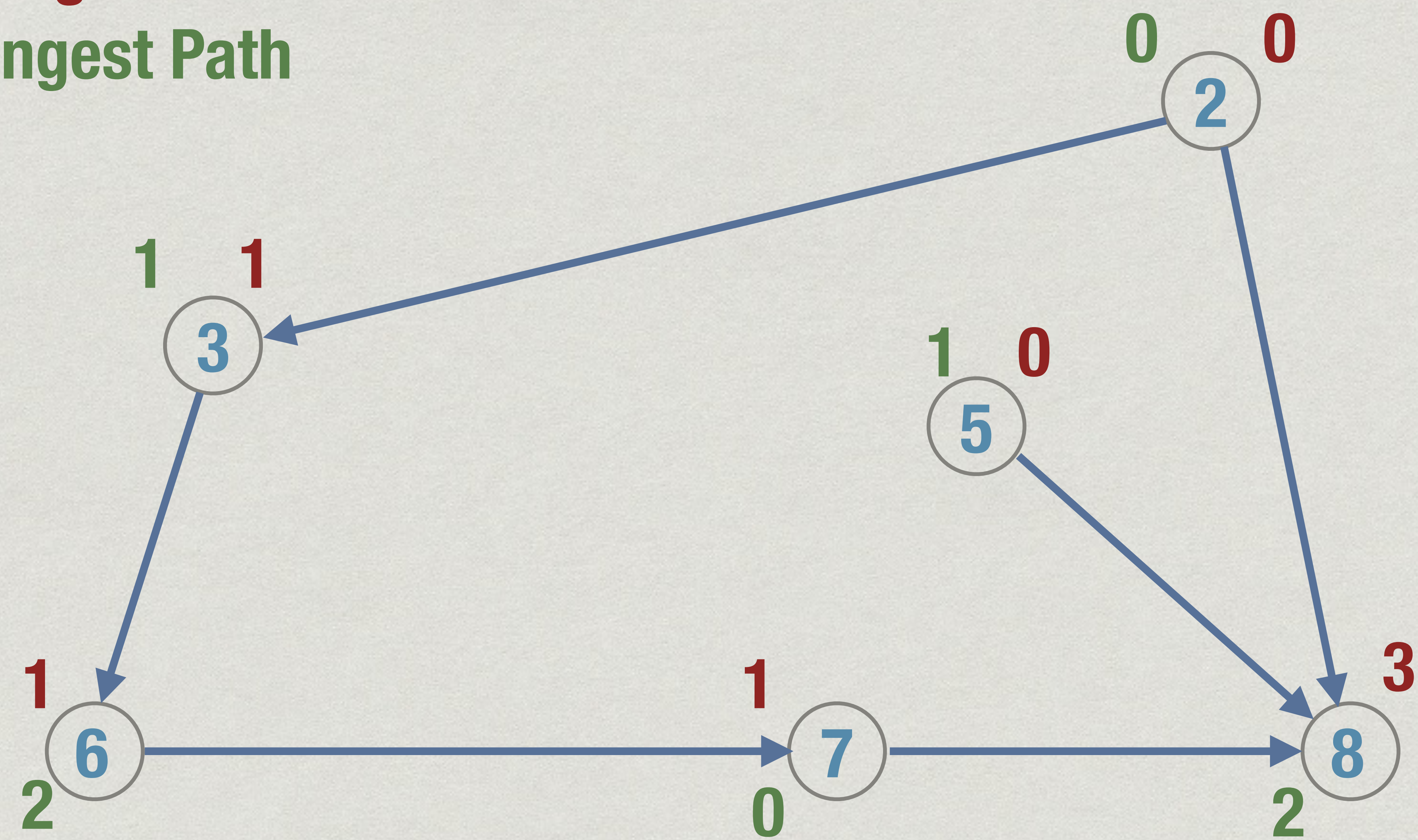
Longest Path



1							
0							

Indegree

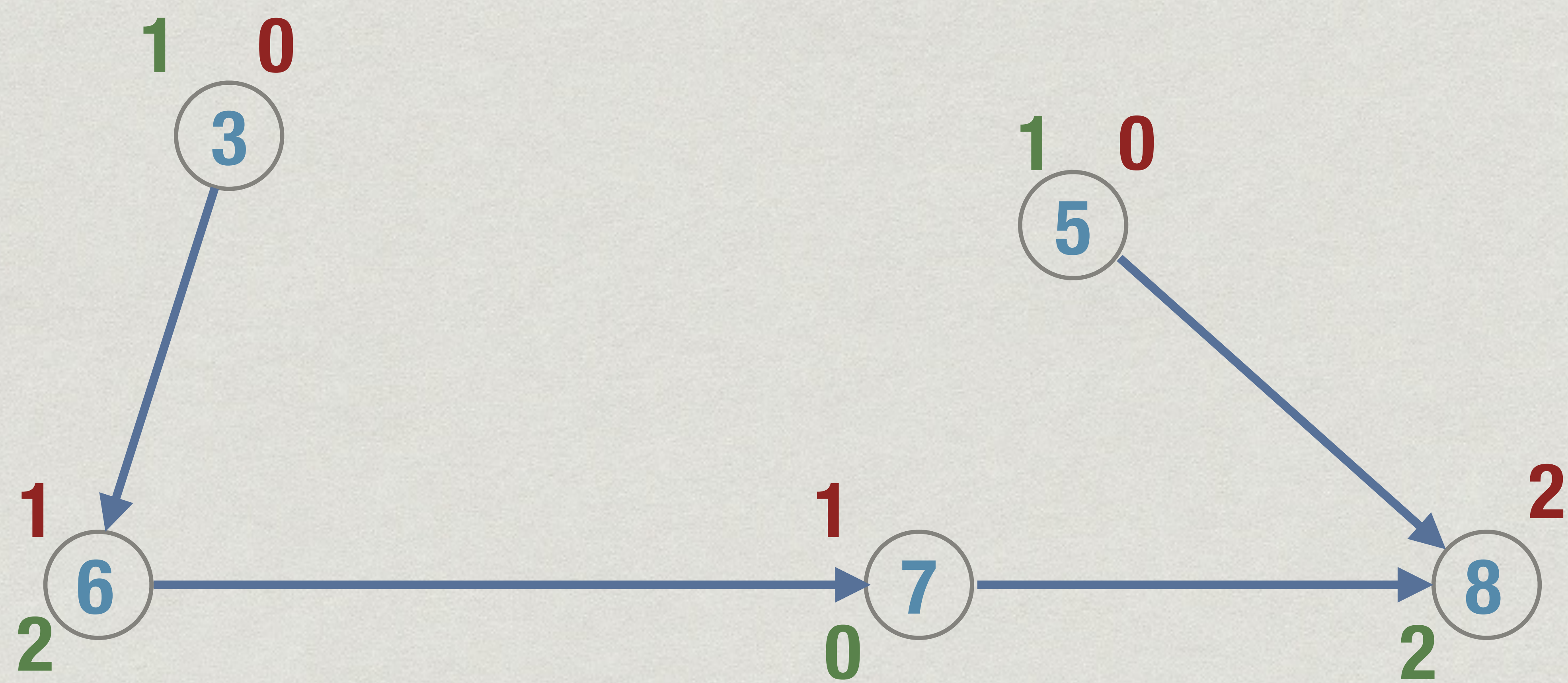
Longest Path



1	4						
0	1						

Indegree

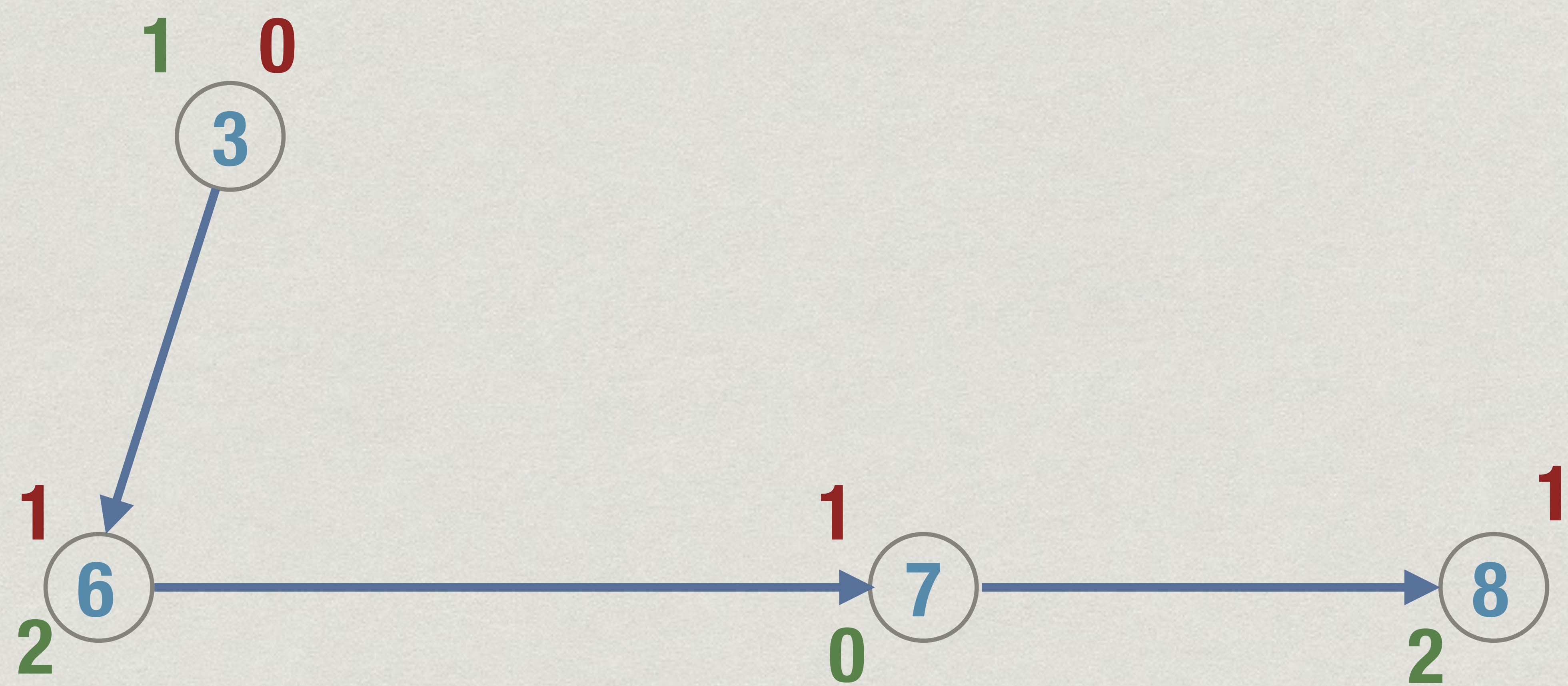
Longest Path



1	4	2					
0	1	0					

Indegree

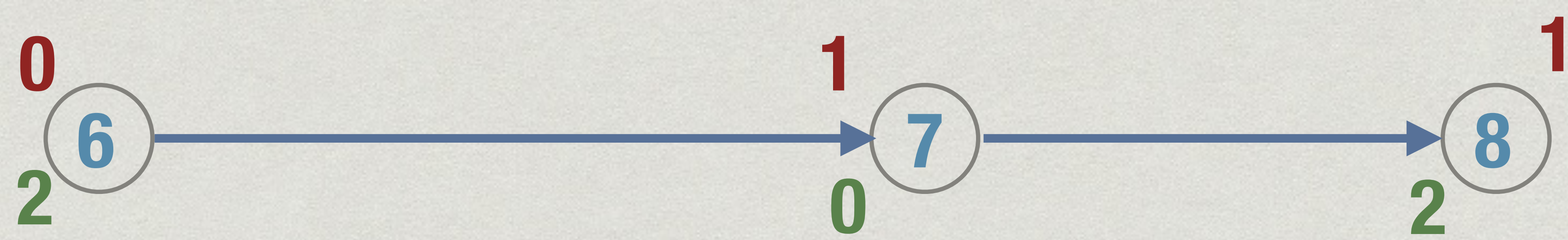
Longest Path



1	4	2	5				
0	1	0	1				

Indegree

Longest Path



1	4	2	5	3			
0	1	0	1	1			

Indegree

Longest Path

1	4	2	5	3	6		
0	1	0	1	1	2		

Indegree

Longest Path

0

8

4

1	4	2	5	3	6	7	
0	1	0	1	1	2	3	

Indegree

Longest Path

1	4	2	5	3	6	7	8
0	1	0	1	1	2	3	4

Topological ordering with longest path

```
function TopologicalOrderWithLongestPath(G)
  for i = 1 to n
    indegree[i] = 0; LPT[i] = 0
    for j = 1 to n
      indegree[i] = indegree[i] + A[j][i]

  for i = 1 to n
    choose j with indegree[j] = 0
      enumerate j
      indegree[j] = -1
      for k = 1 to n
        if A[j][k] == 1
          indegree[k] = indegree[k]-1
          LPT[k] = max(LPT[k], 1 + LPT[j])
```

Topological ordering with longest path

- This implementation has complexity is $O(n^2)$
- As before, we can use adjacency lists to improve the complexity to $O(m+n)$

Topological ordering with longest path 2

```
function TopologicalOrder(G) //Edges are in adjacency list
  for i = 1 to n { indegree[i] = 0; LPT[i] = 0}

  for i = 1 to n
    for (i,j) in E //proportional to outdegree(i)
      indegree[j] = indegree[j] + 1

  for i = 1 to n
    if indegree[i] == 0 { add i to Queue }

  while Queue is not empty
    j = remove_head(Queue)
    for (j,k) in E //proportional to outdegree(j)
      indegree[k] = indegree[k] - 1
      LPT[k] = max(LPT[k], 1 + LPT[j])
      if indegree[k] == 0 { add k to Queue }
```

Summary

- Dependencies are naturally modelled using DAGs
- Topological ordering lists vertices without violating dependencies
- Longest path in a DAG represents minimum number of steps to list all vertices in groups