Applications of Flows + Cuts

Applications of Flows + Cuts

(high-level comment)

"only two algorithms"*

① identify subproblems

② change the input

Applications of Flows + Cuts

Gameplan

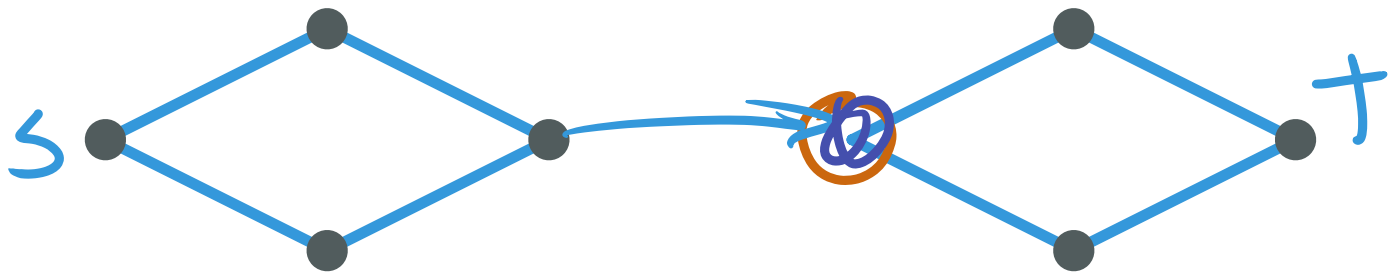
1. Survey a bunch of problems
2. Start solving them one by one

"only two algorithms"*

① identify subproblems

② change the input

Vertex Disjoint Paths



2 edge disjoint paths

1 vertex disjoint path

vertex cut = 1

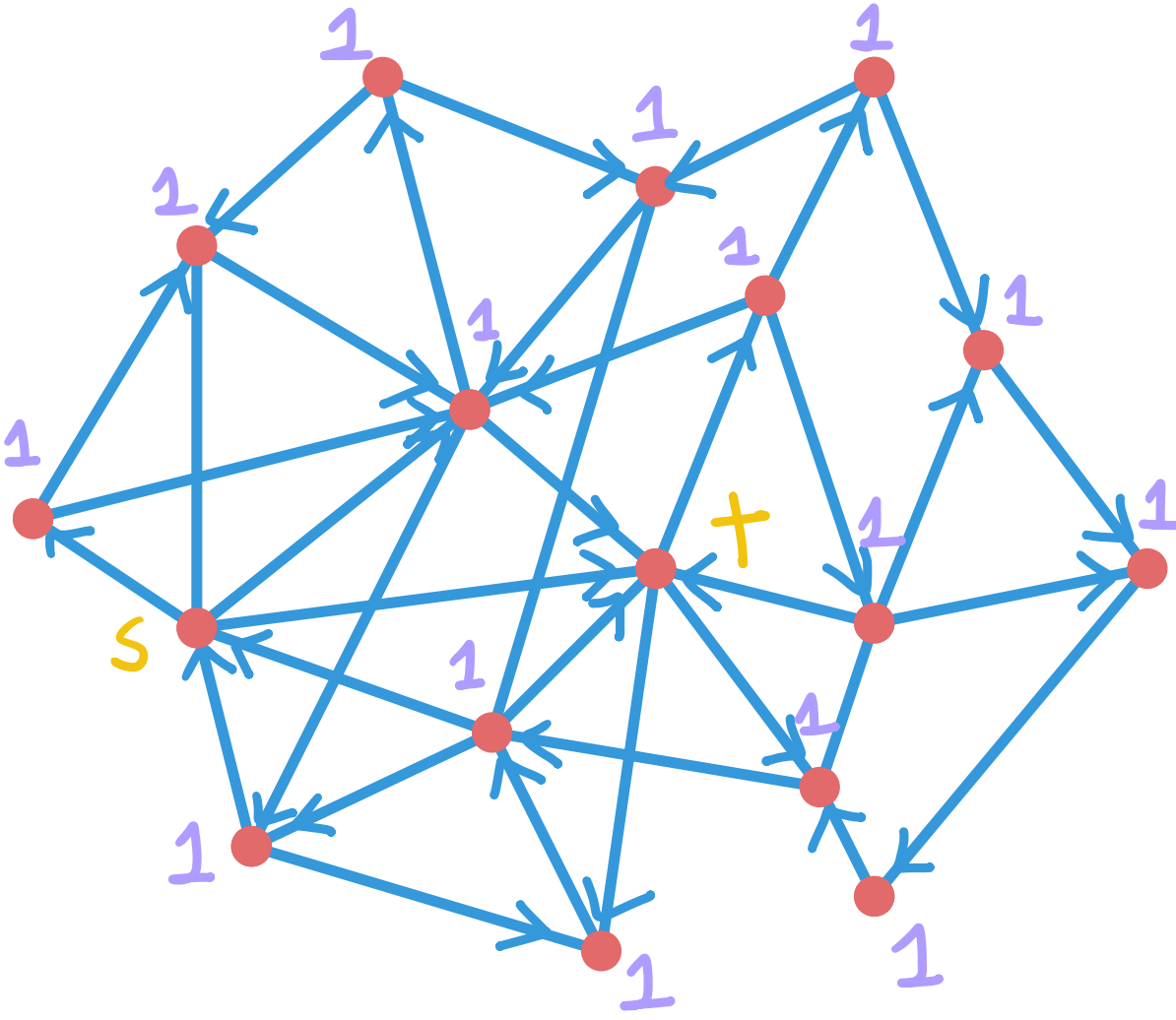
Vertex capacities $c : V \to \mathbb{R}_{>0}$

$c(v)$ = limit of flow through v

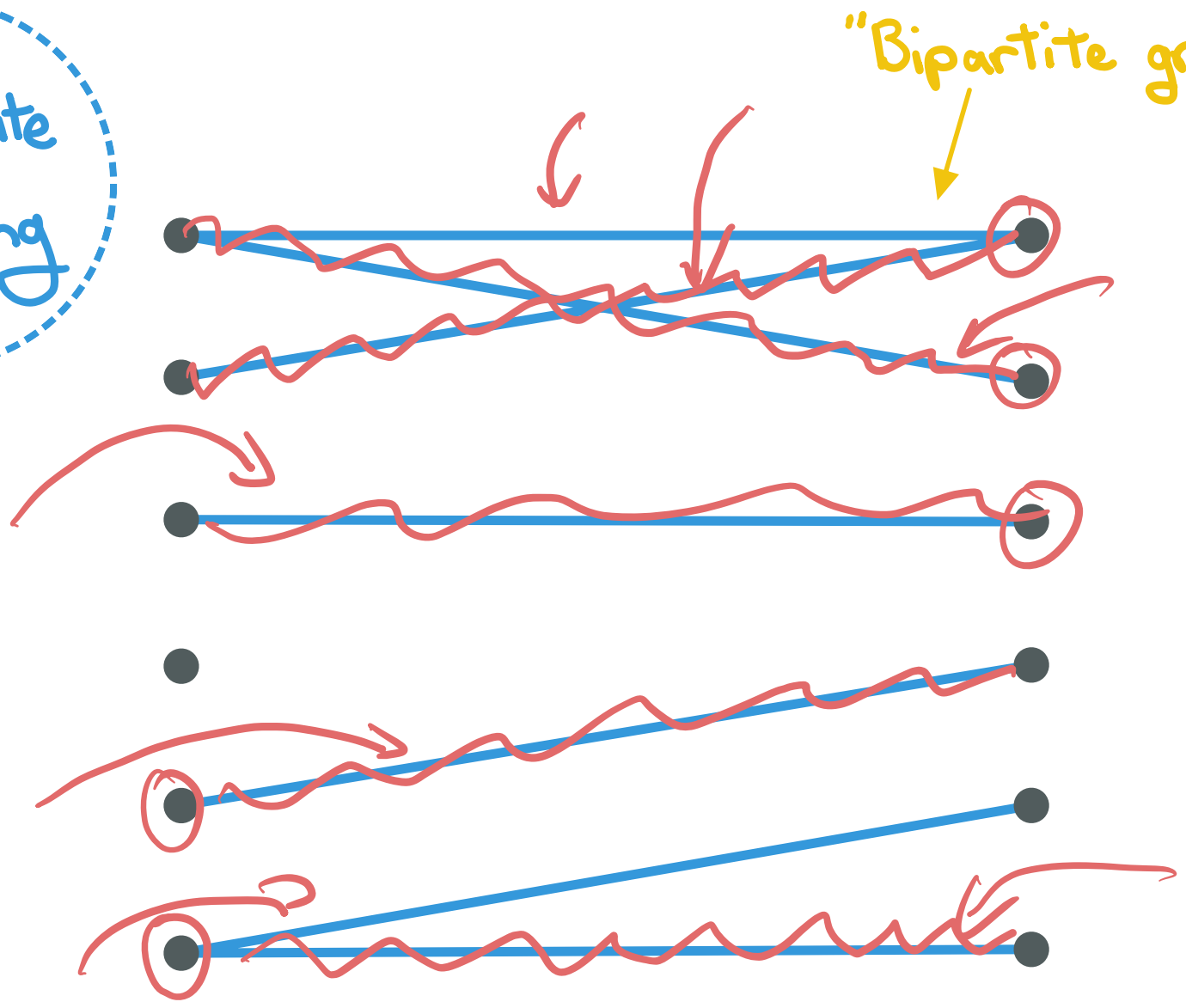
e.g. $c(v) = 1$

$\forall v \notin \{s, t\}$

=> vertex disjoint paths



Bipartite Matching

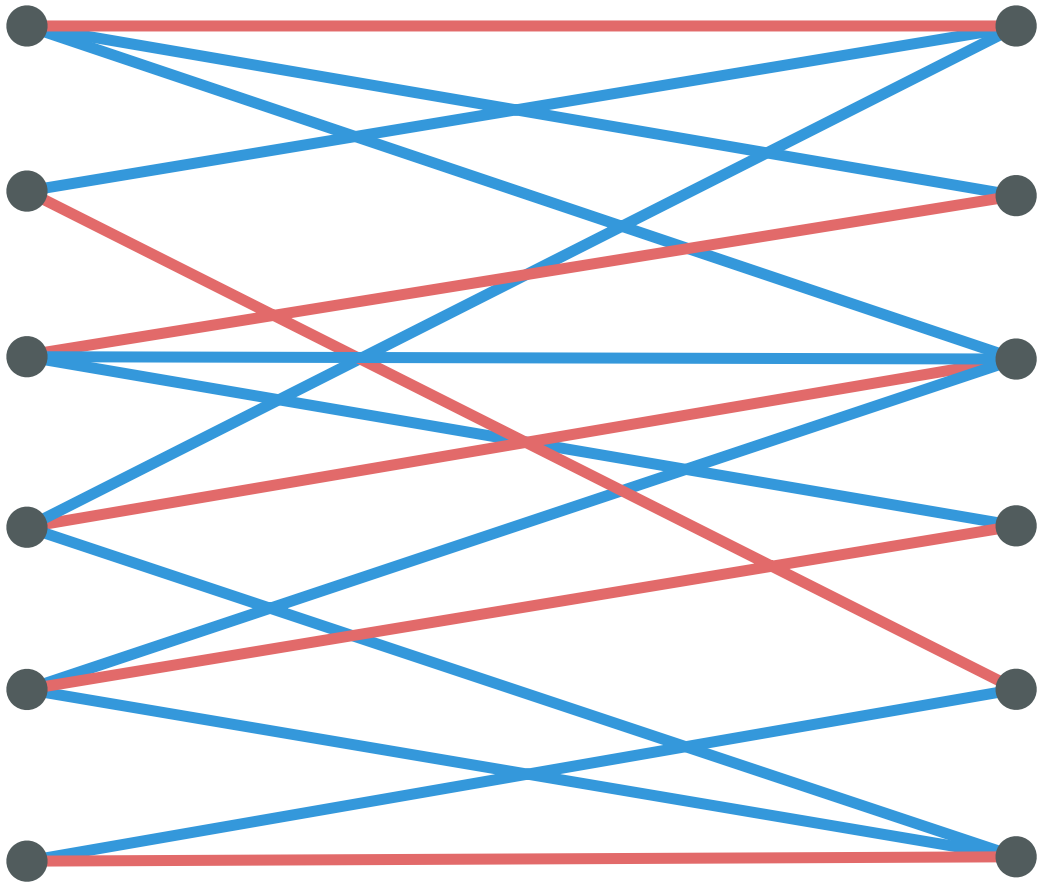


Goal: max set of edges w/ distinct endpoints

"matching"

Bipartite Matching

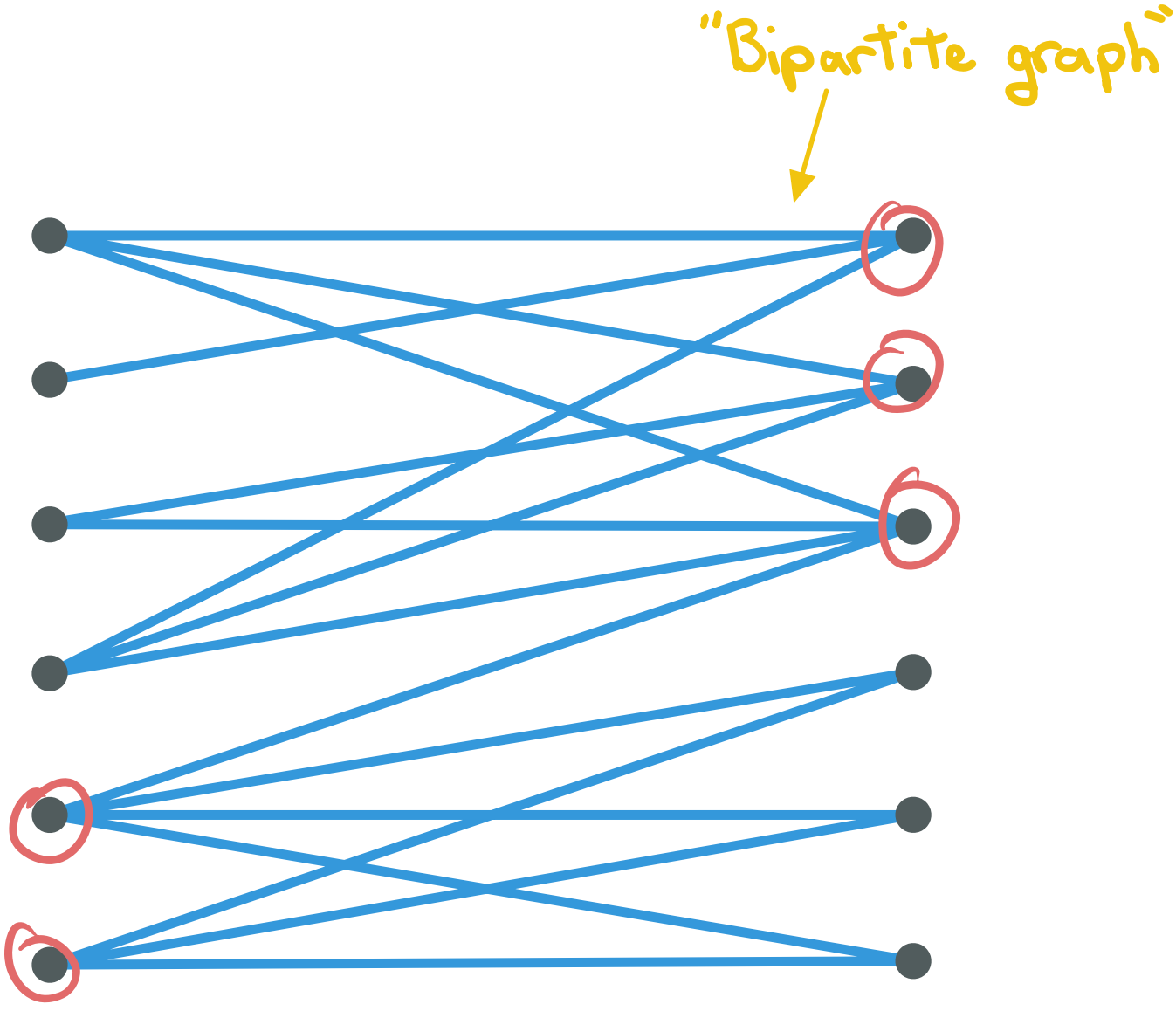
(bipartite graph)



Goal: max set of edges w/ distinct endpoints

"matching"

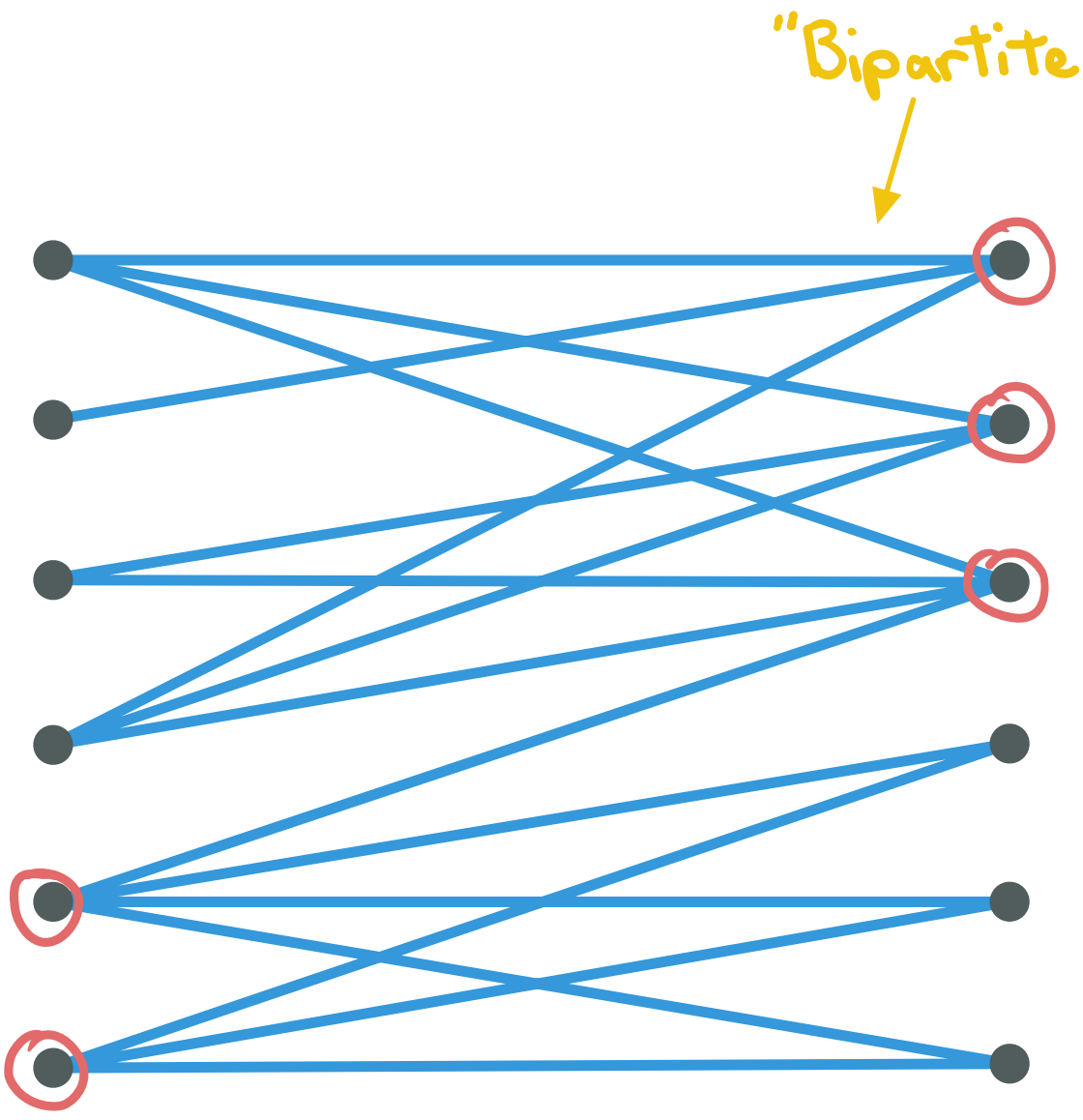
Bipartite Vertex Cover



Goal: min set of vertices incident to all edges

"vertex cover"

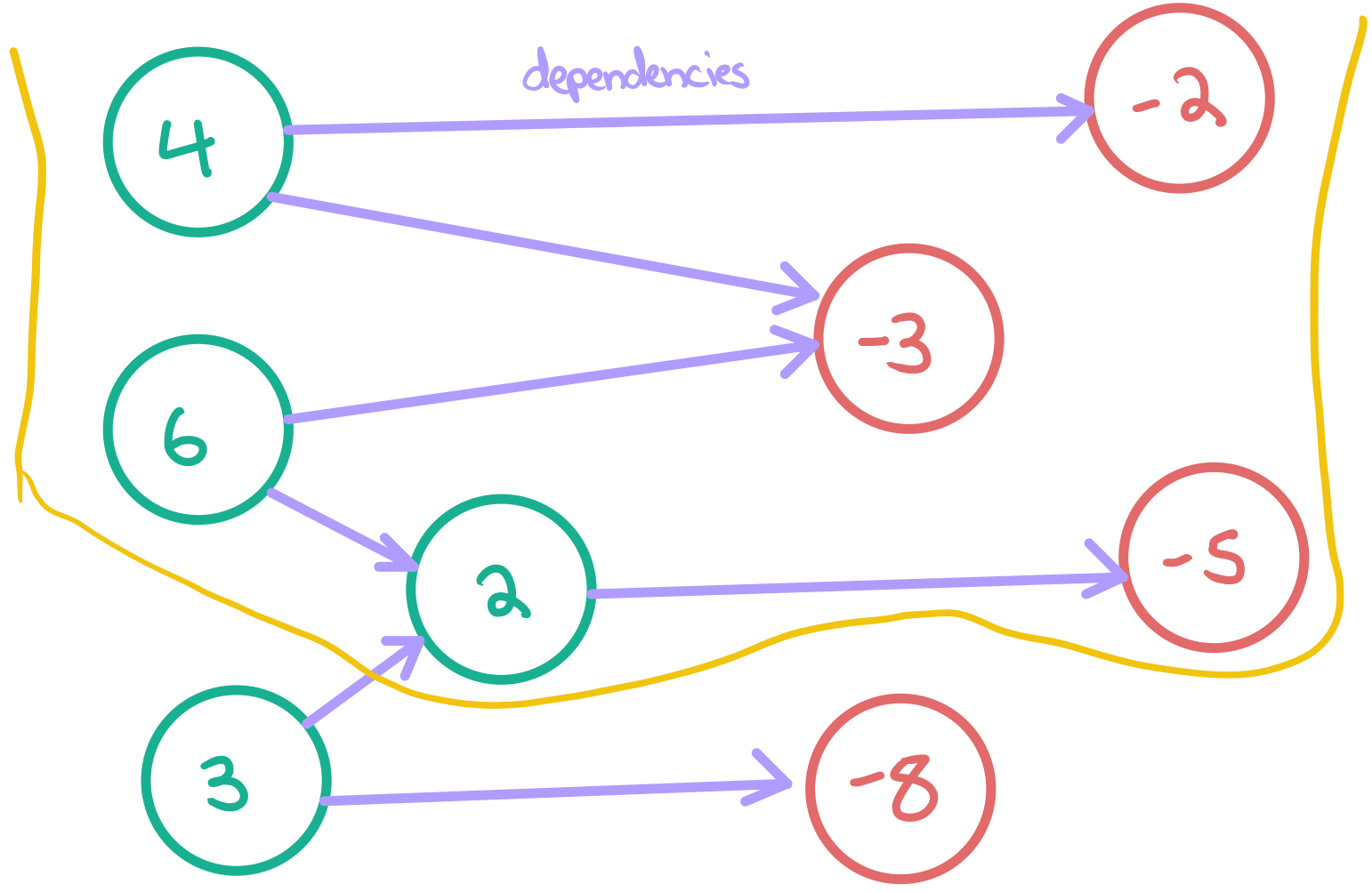
Bipartite Vertex Cover



Goal: min set of vertices incident to all edges

"vertex cover"

Project Selection



Projects have net profits (can be <0 for costs) and dependencies

Goal: select projects (including dependencies) w/ max total profit

Playoff Elimination

Playoff Elimination

can Boston catch the Yankees?

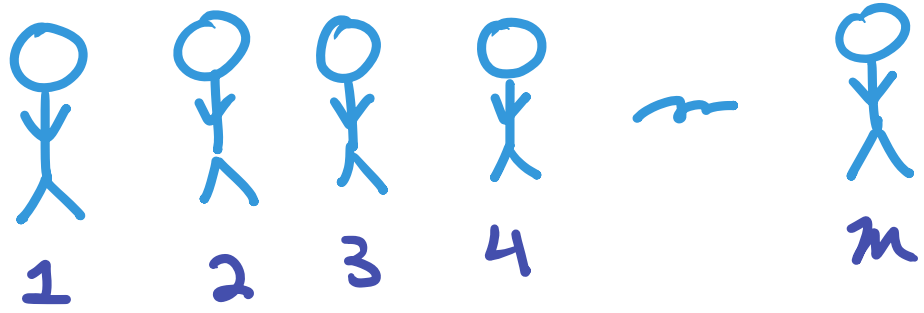
Team	Wins	Games left
New York	92	2
Baltimore	91	3
Toronto	91	3
Boston	90	2

Remaining schedule

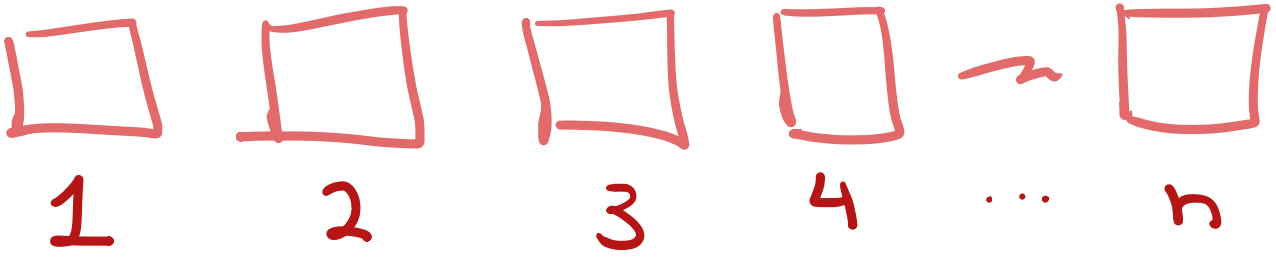
(wins)	New York (92)	Baltimore (91)	Toronto (91)	Boston (90)
New York	×	1	1	0
Baltimore	1	×	1	1
Toronto	1	1	×	1
Boston	0	1	1	×

Assignment Problems

m individuals



n jobs

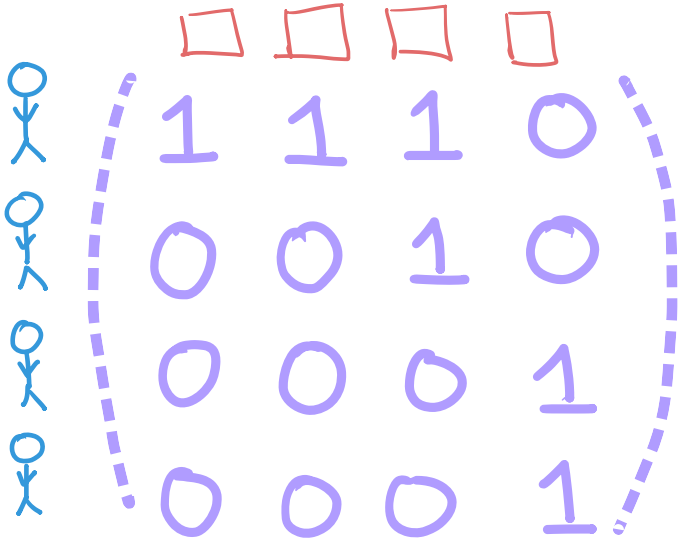


goal: fill as many jobs as possible w/ individuals

the catch: not everyone is qualified for all jobs

Assignment Problems

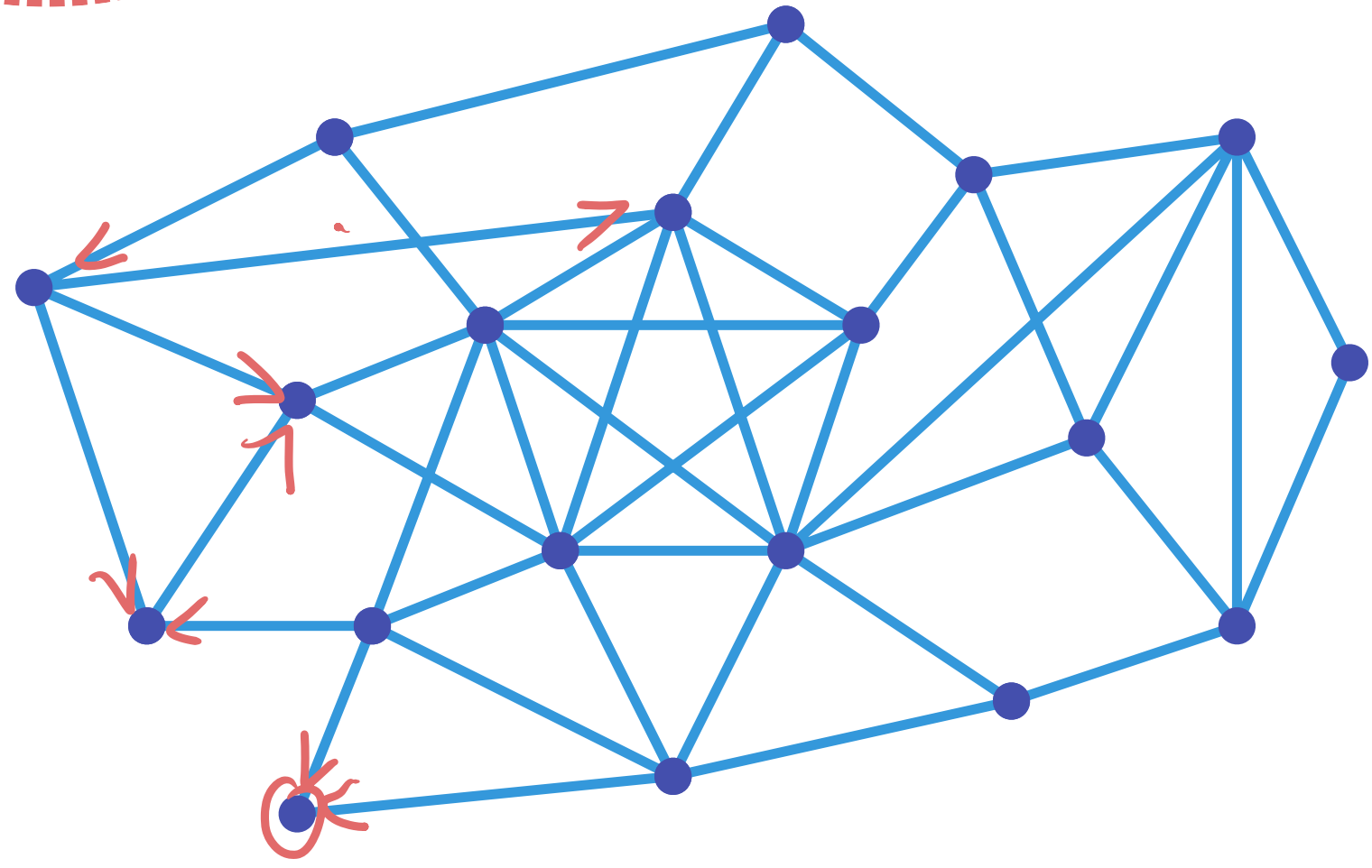
"Qualification matrix"



"1" = qualified "0" = not qualified

Edge Orientation

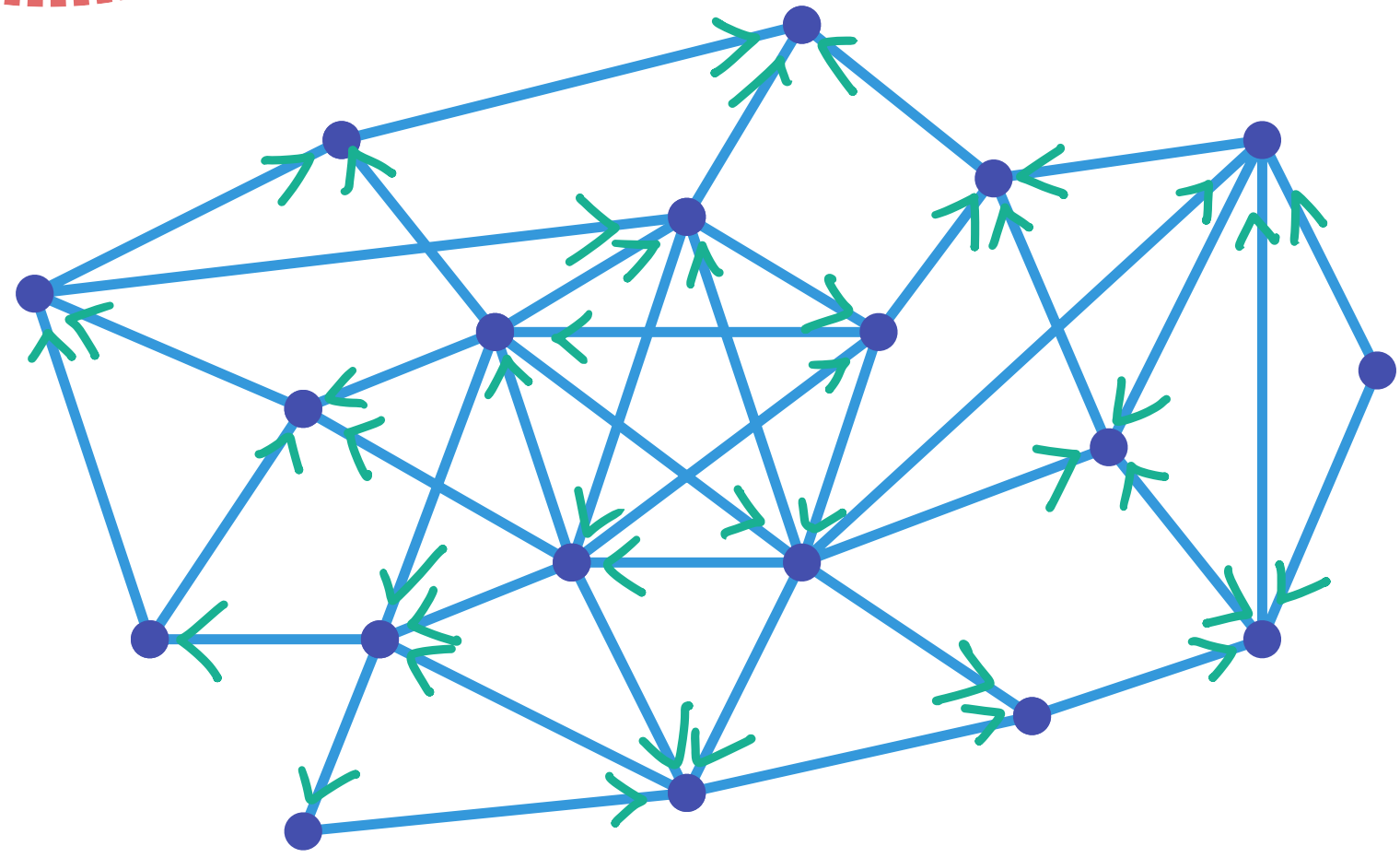
input: undirected graph



Goal: direct the edges to minimize the max in-degree

Edge Orientation

input: undirected graph

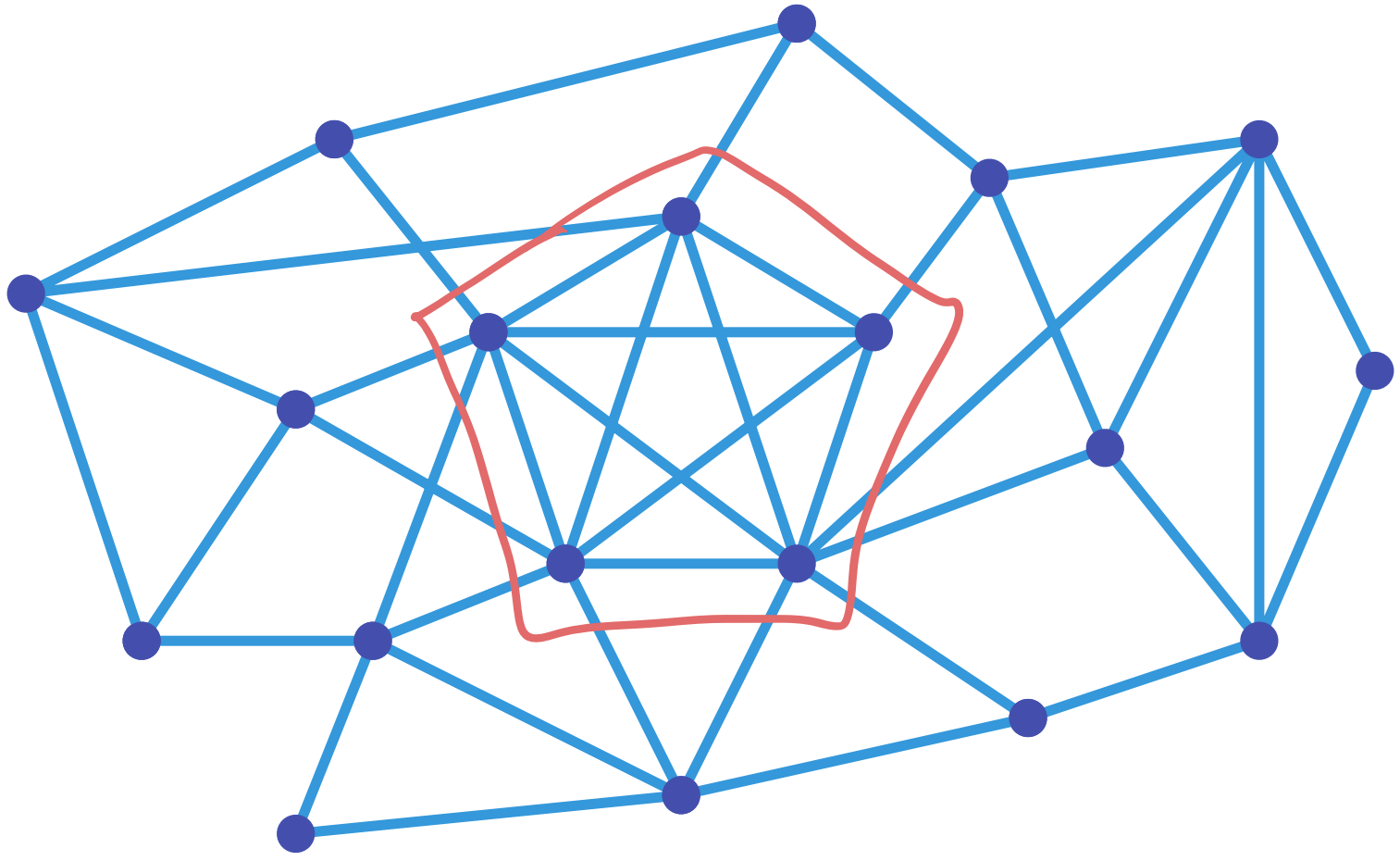


Goal: direct the edges to minimize the max in-degree

Densest Subgraph

"density" = $\frac{\#\text{ edges}}{\#\text{ vertices}}$

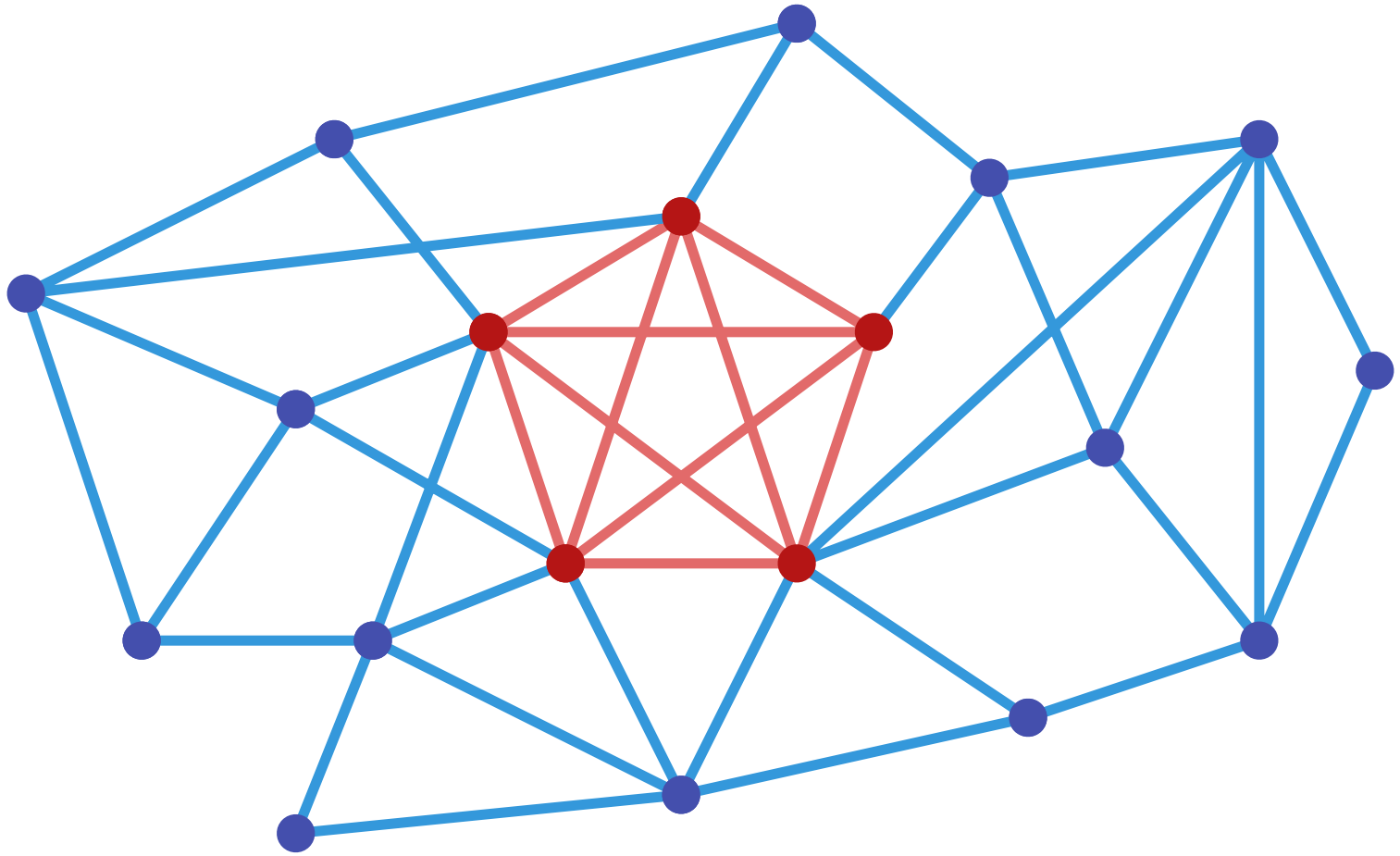
$\frac{\binom{5}{2}}{5}$



Goal: compute densest subgraph

Densest Subgraph

"density" = $\frac{\#\text{ edges}}{\#\text{ vertices}}$

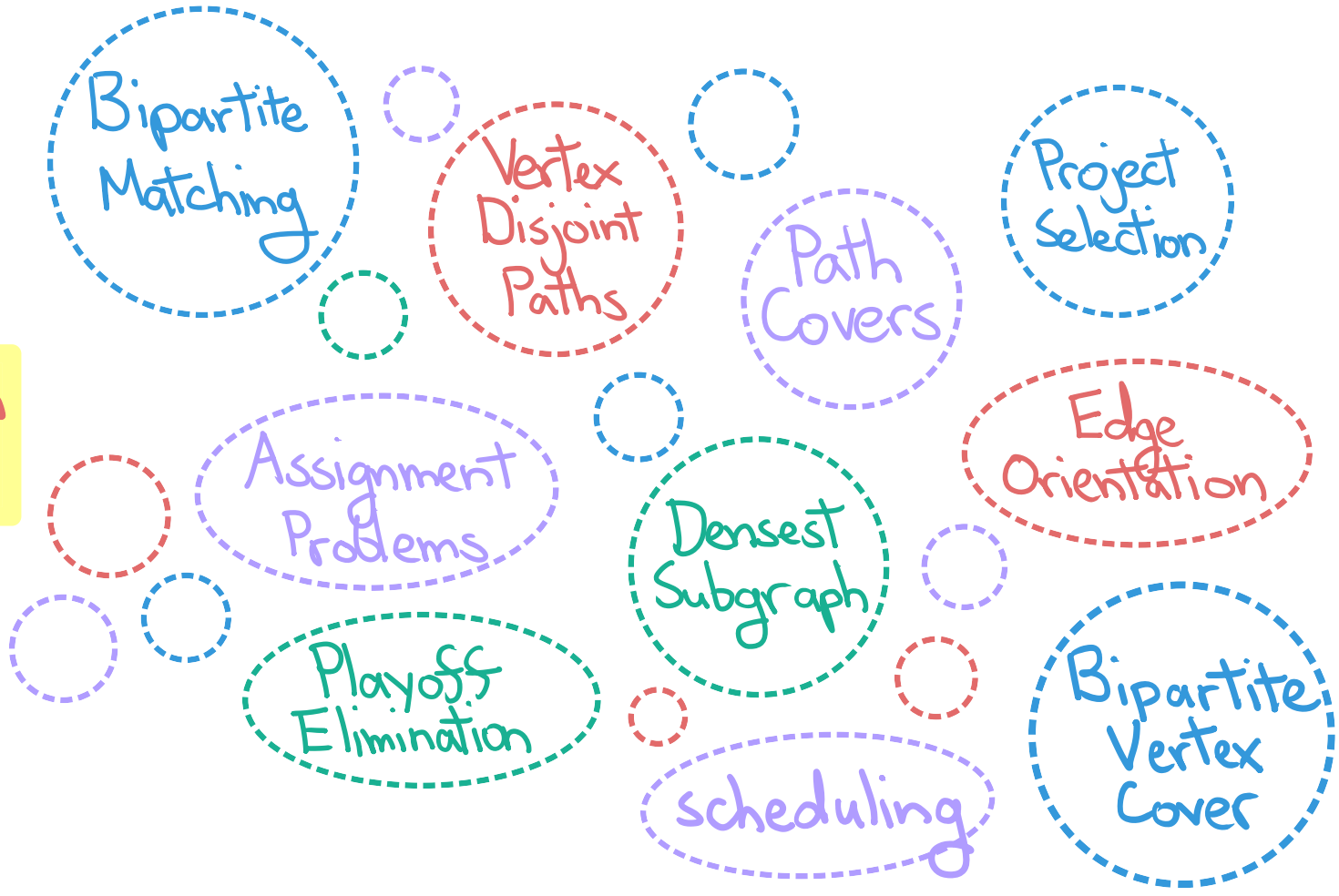


Goal: compute densest subgraph

Applications of Flows + Cuts

Gameplan

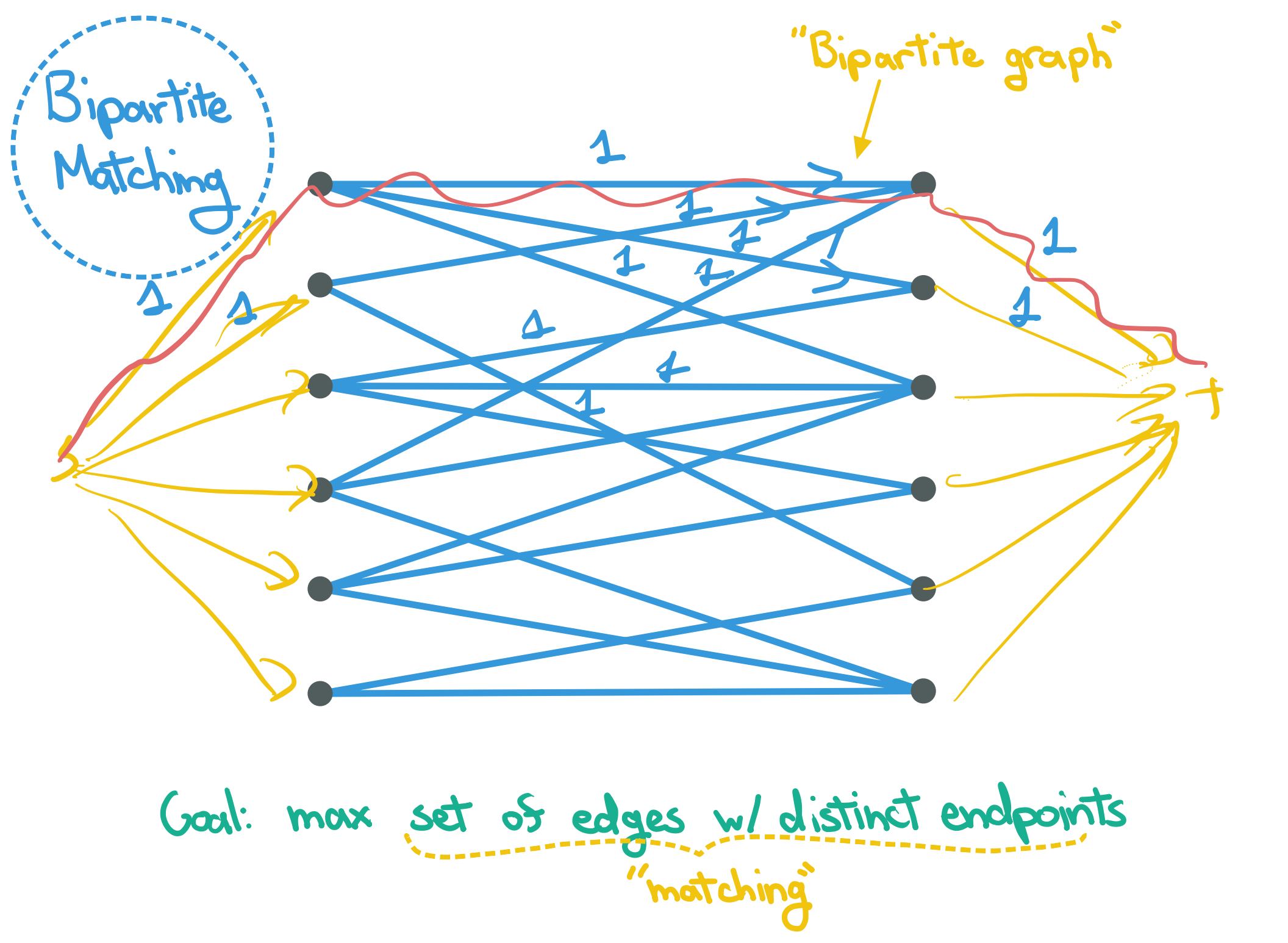
1. Survey a bunch of problems
2. Start solving them one by one



"only two algorithms"*

① identify subproblems

② change the input

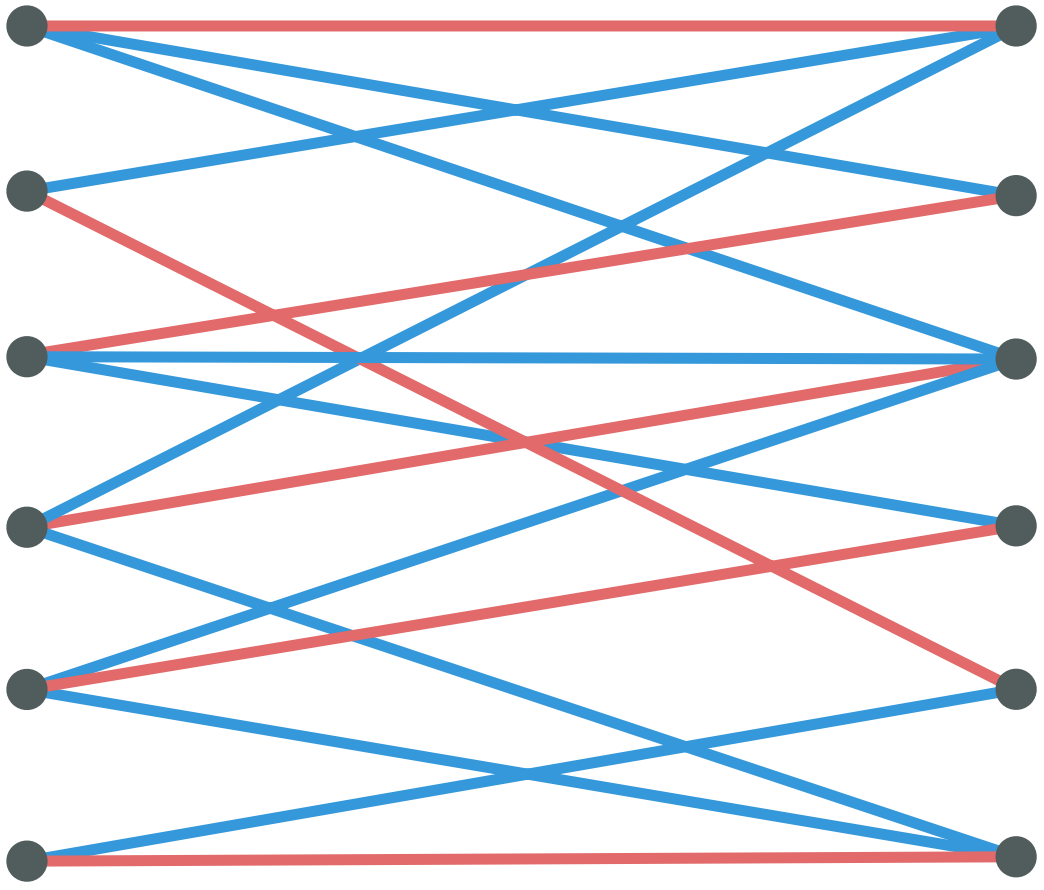


Goal: max set of edges w/ distinct endpoints

"matching"

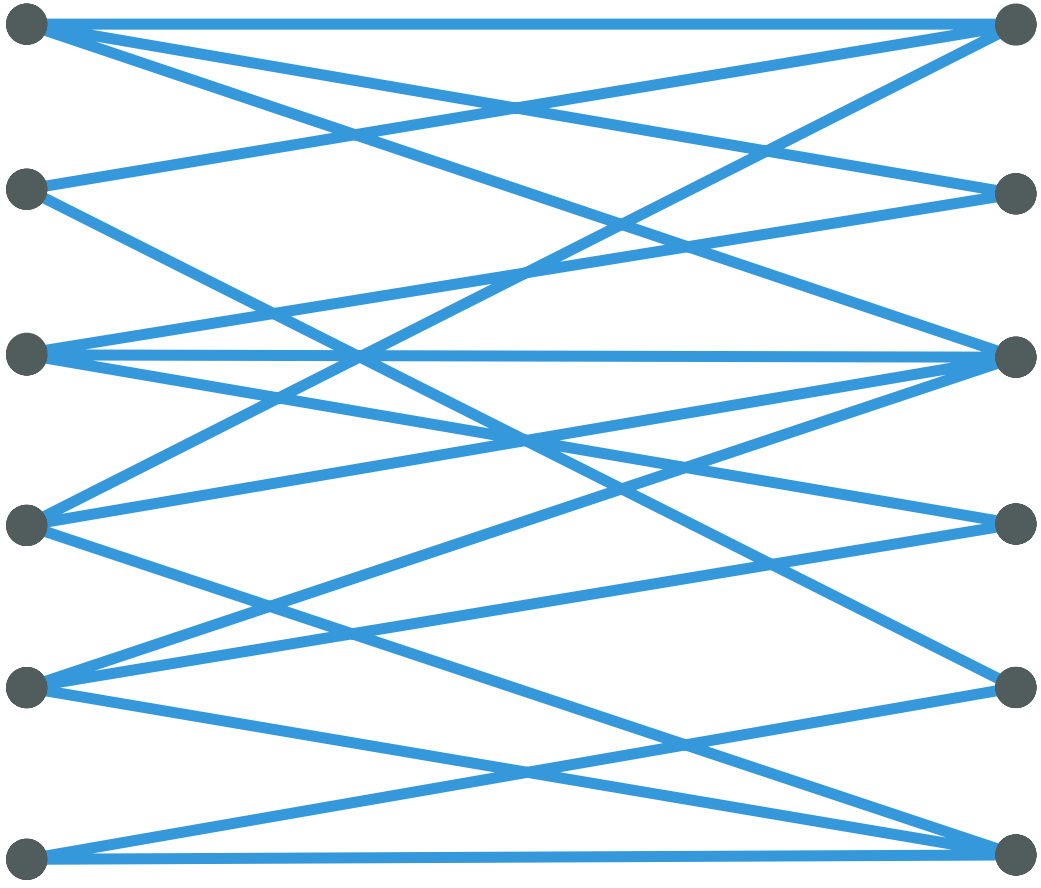
Bipartite Matching

(bipartite graph)

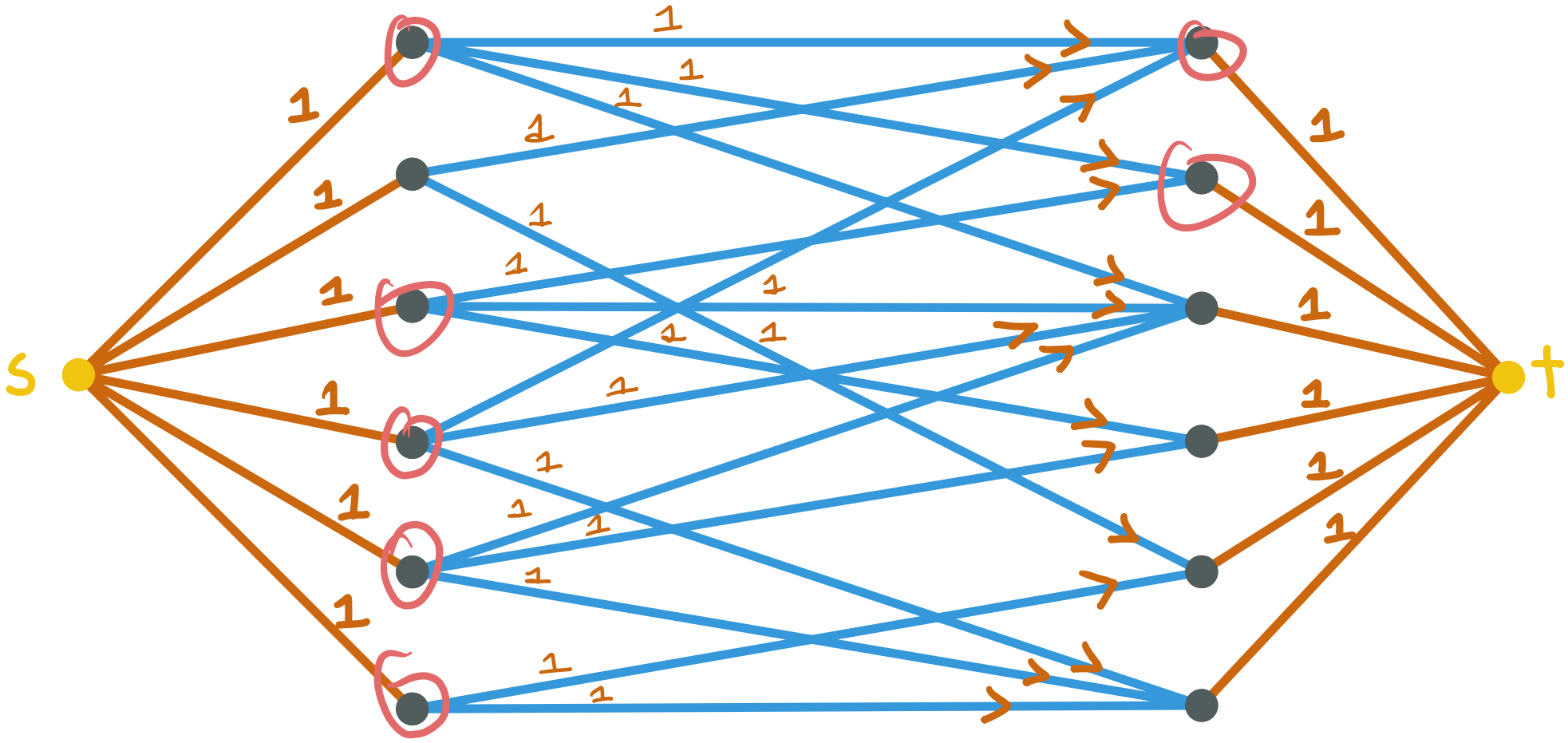


Goal: max set of edges w/ distinct endpoints

"matching"



(see lecture notes for detailed proof)



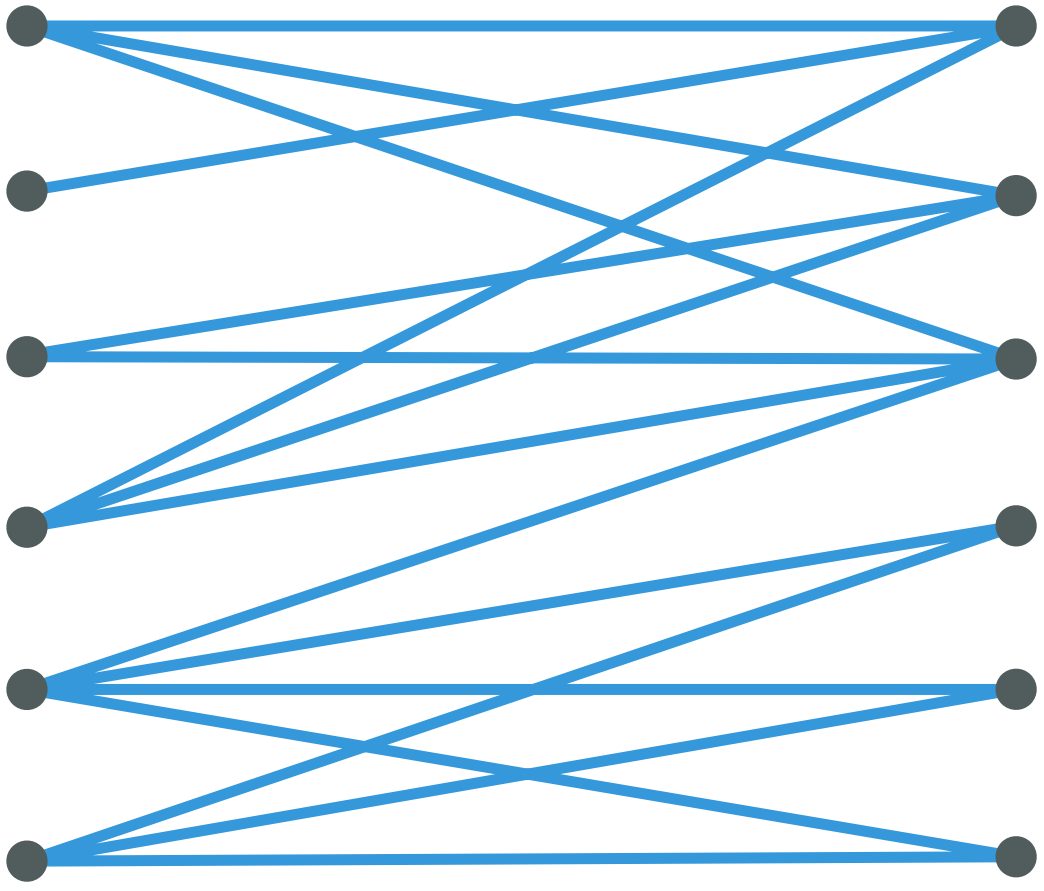
Matching w/ k edges

k edge-disjoint (s,t)-paths

(see lecture notes for detailed proof)

Bipartite Vertex Cover

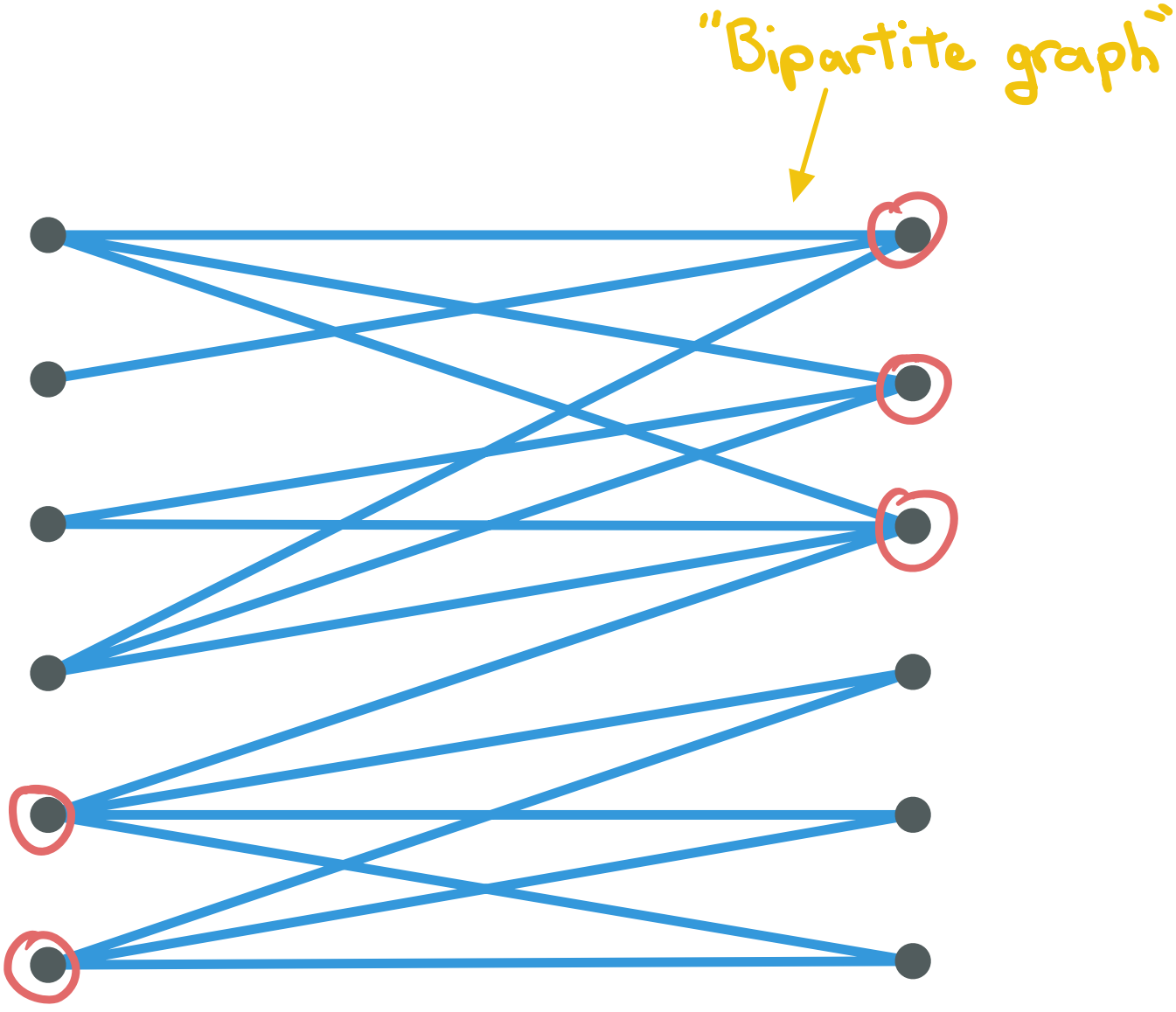
"Bipartite graph"



Goal: min set of vertices incident to all edges

"vertex cover"

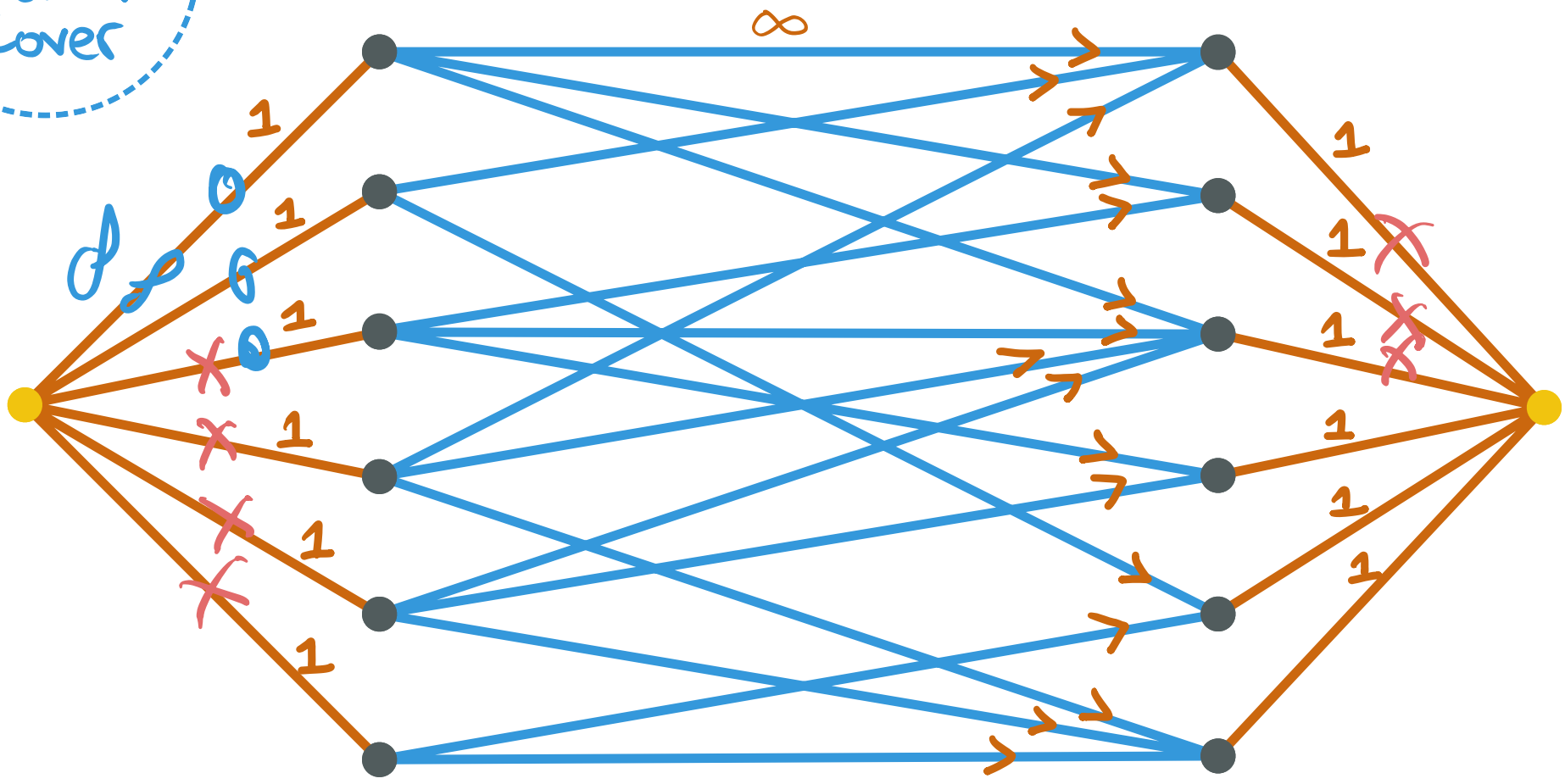
Bipartite Vertex Cover



Goal: min set of vertices incident to all edges

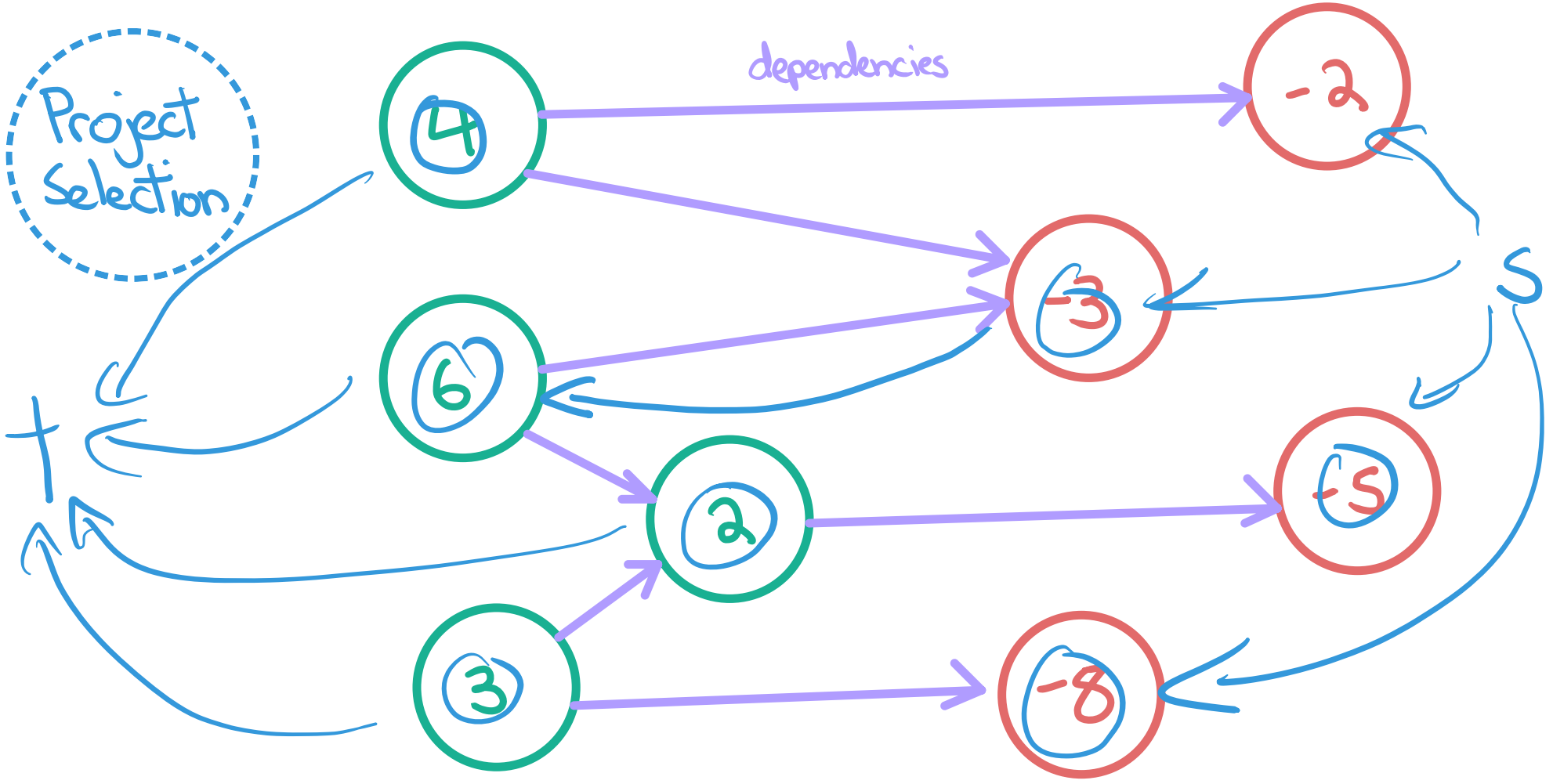
"vertex cover"

Bipartite Vertex Cover



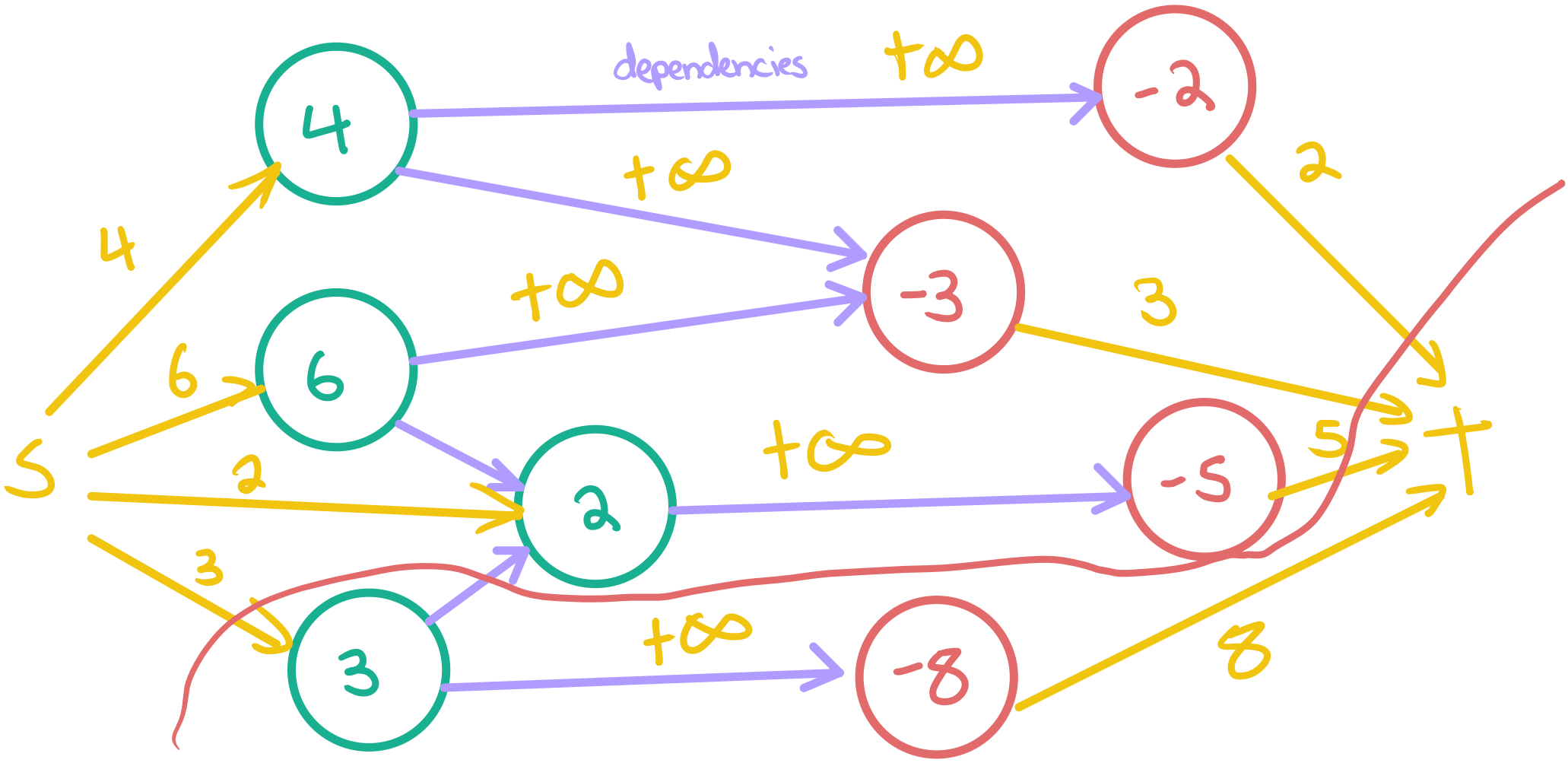
(s,t)-cut of size k

vertex cover of size k

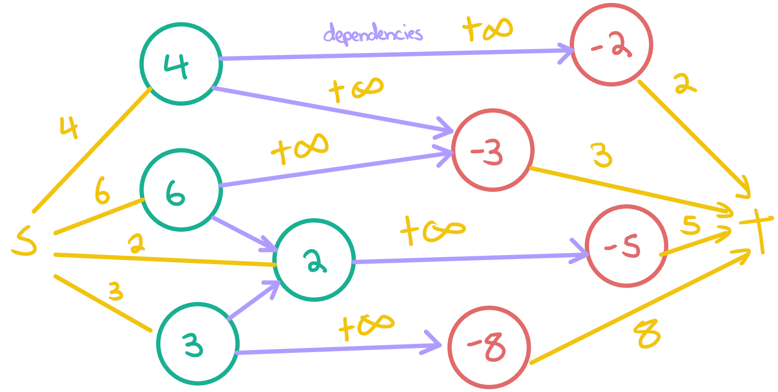


Projects have net profits (can be <0 for costs) and dependencies

Goal: select projects (including dependencies) w/ max total profit



claim: min-cut => best project selection


dependencies
+∞
+∞
+∞
+∞
+∞
4
6
2
3
-2
-3
-5
-8
s
t
5
8

Playoff Elimination

Playoff Elimination

can Boston catch the Yankees?

Team	Wins	Games left
New York	92	2
Baltimore	91	3
Toronto	91	3
Boston	90	2

Remaining schedule

(wins)	(92) New York	(91) Baltimore	(91) Toronto	(90) Boston
New York	X	1	1	0
Baltimore	1	X	1	1
Toronto	1	1	X	1
Boston	0	1	1	X

(wins)	(92) NewYork	(91) Baltimore	(91) Toronto	(90) Boston
NewYork	X	1	1	0
Baltimore	1	X	1	1
Toronto	1	1	X	1
Boston	0	1	1	X

NewYork vs Baltimore

NewYork vs Toronto

Baltimore vs Toronto

Baltimore vs Boston

Toronto vs Boston

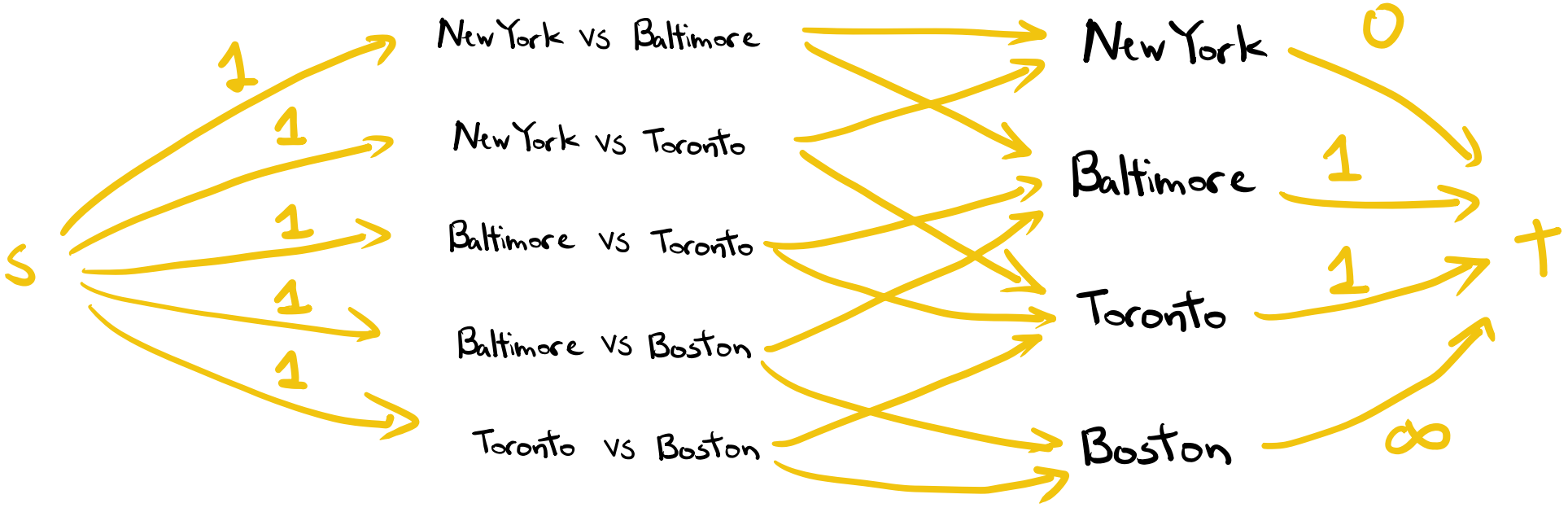
NewYork

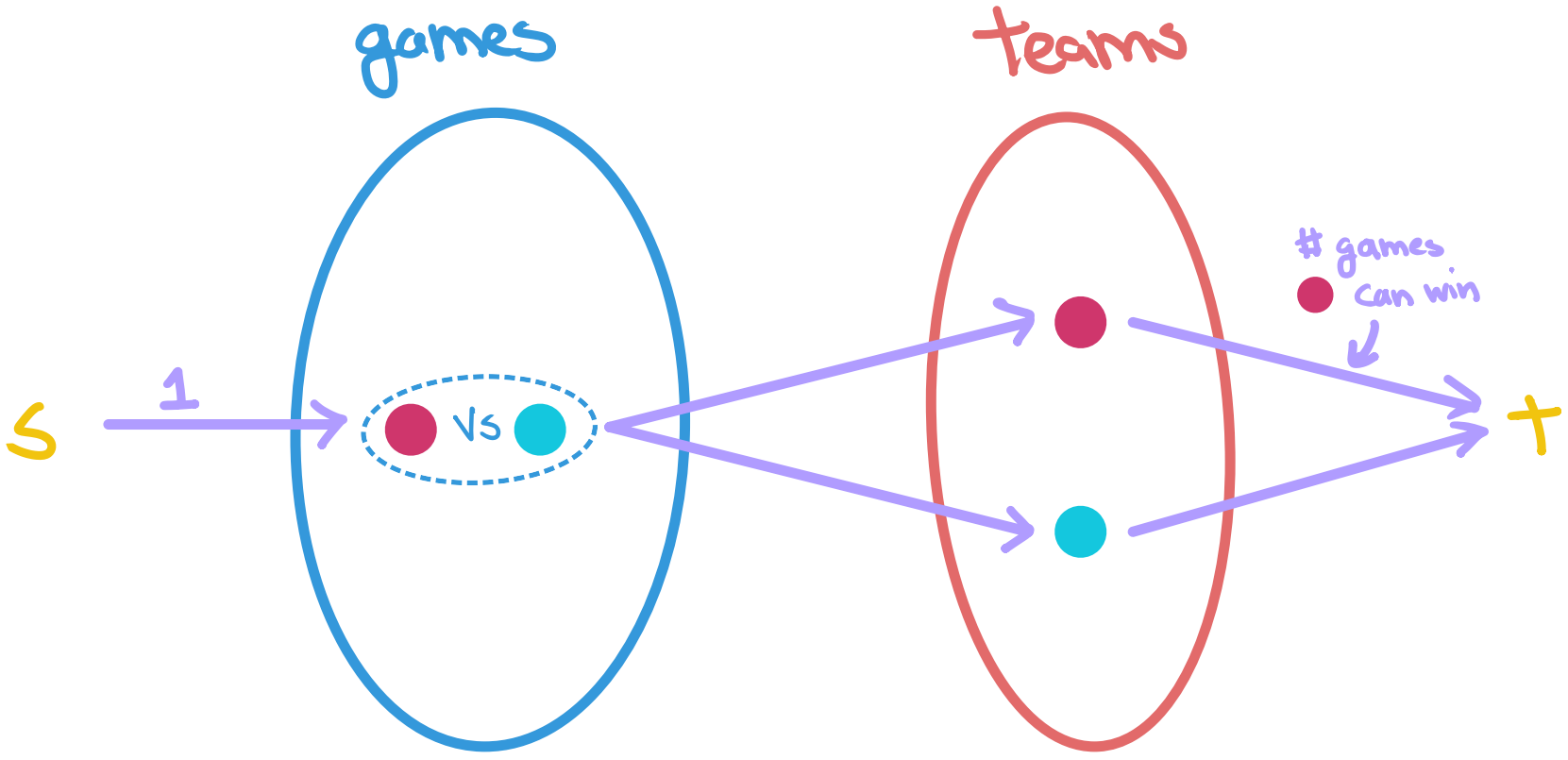
Baltimore

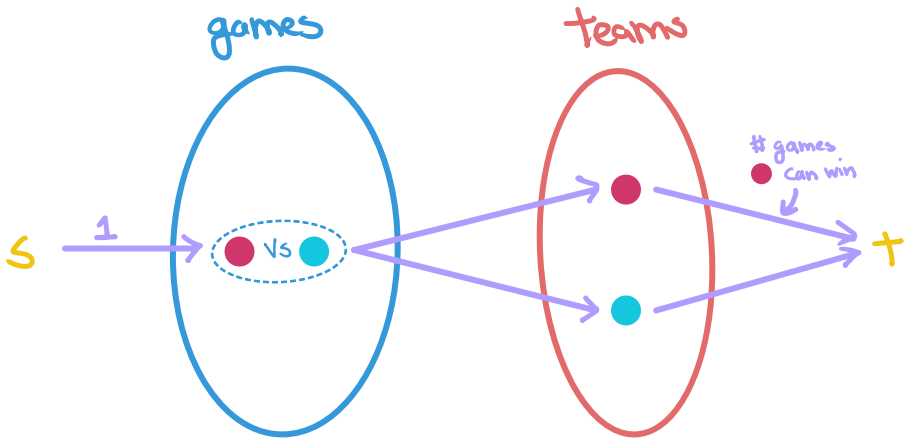
Toronto

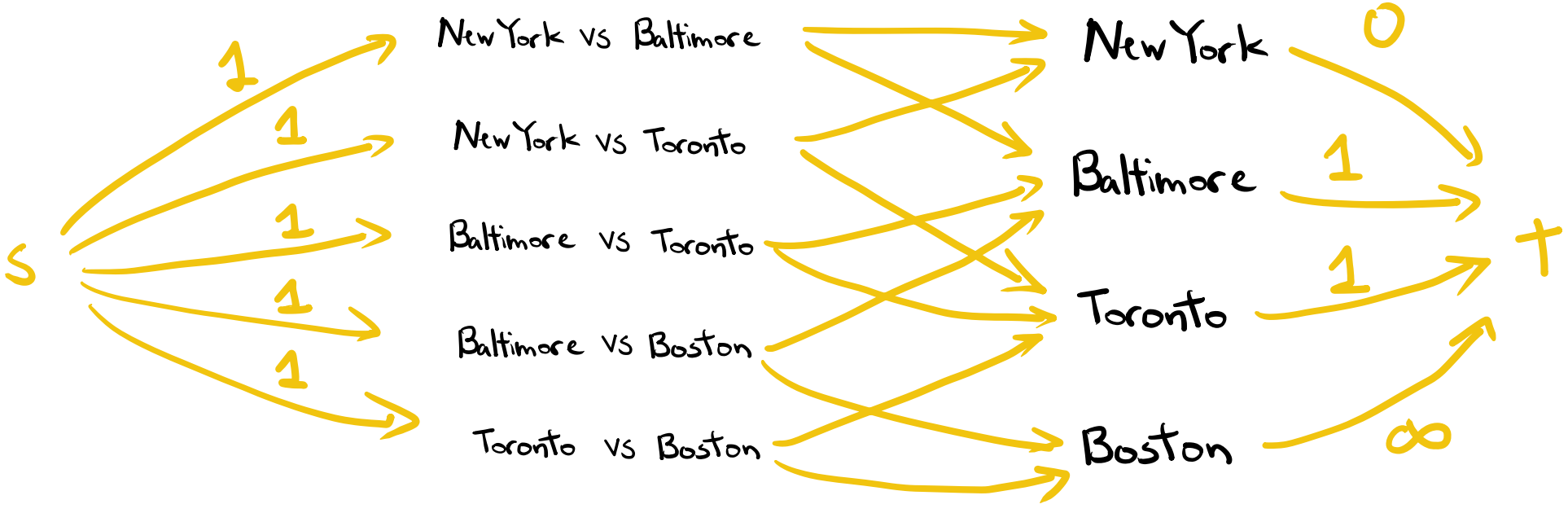
Boston

(wins)	(92) NewYork	(91) Baltimore	(91) Toronto	(90) Boston
NewYork	X	1	1	0
Baltimore	1	X	1	1
Toronto	1	1	X	1
Boston	0	1	1	X




games
teams
games
can win
s
1
vs
t

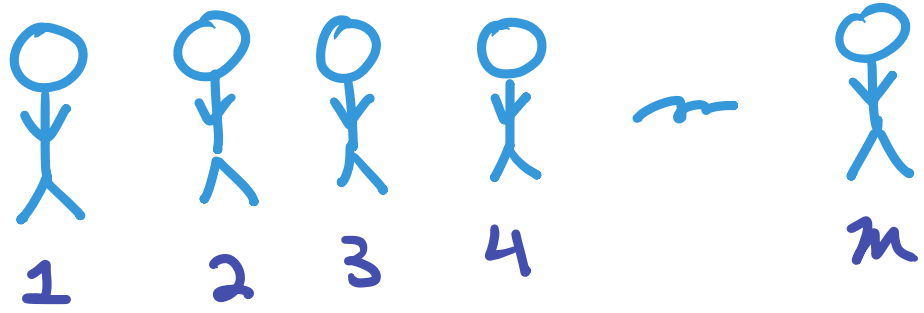




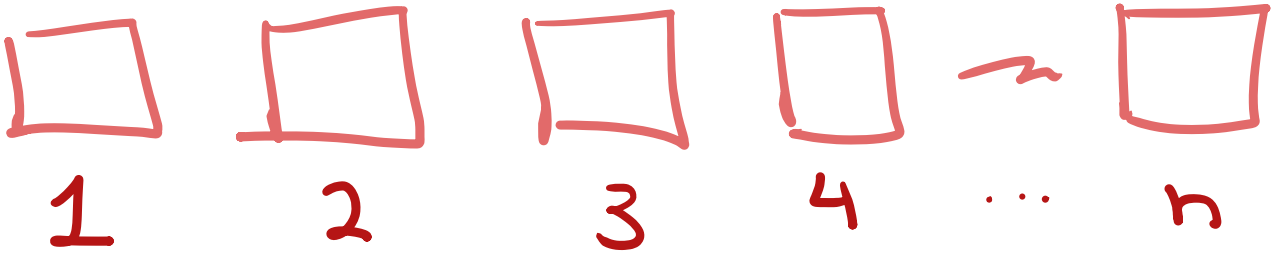
"allocating wins"

Assignment Problems

m individuals



n jobs

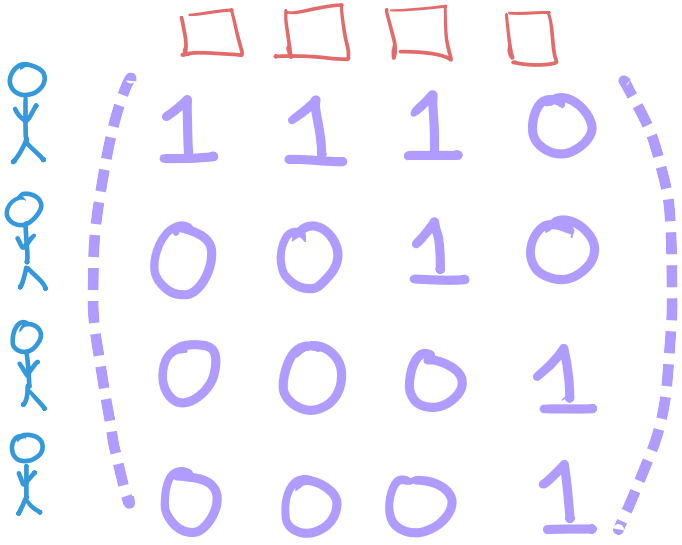


goal: fill as many jobs as possible w/ individuals

the catch: not everyone is qualified for all jobs

Assignment Problems

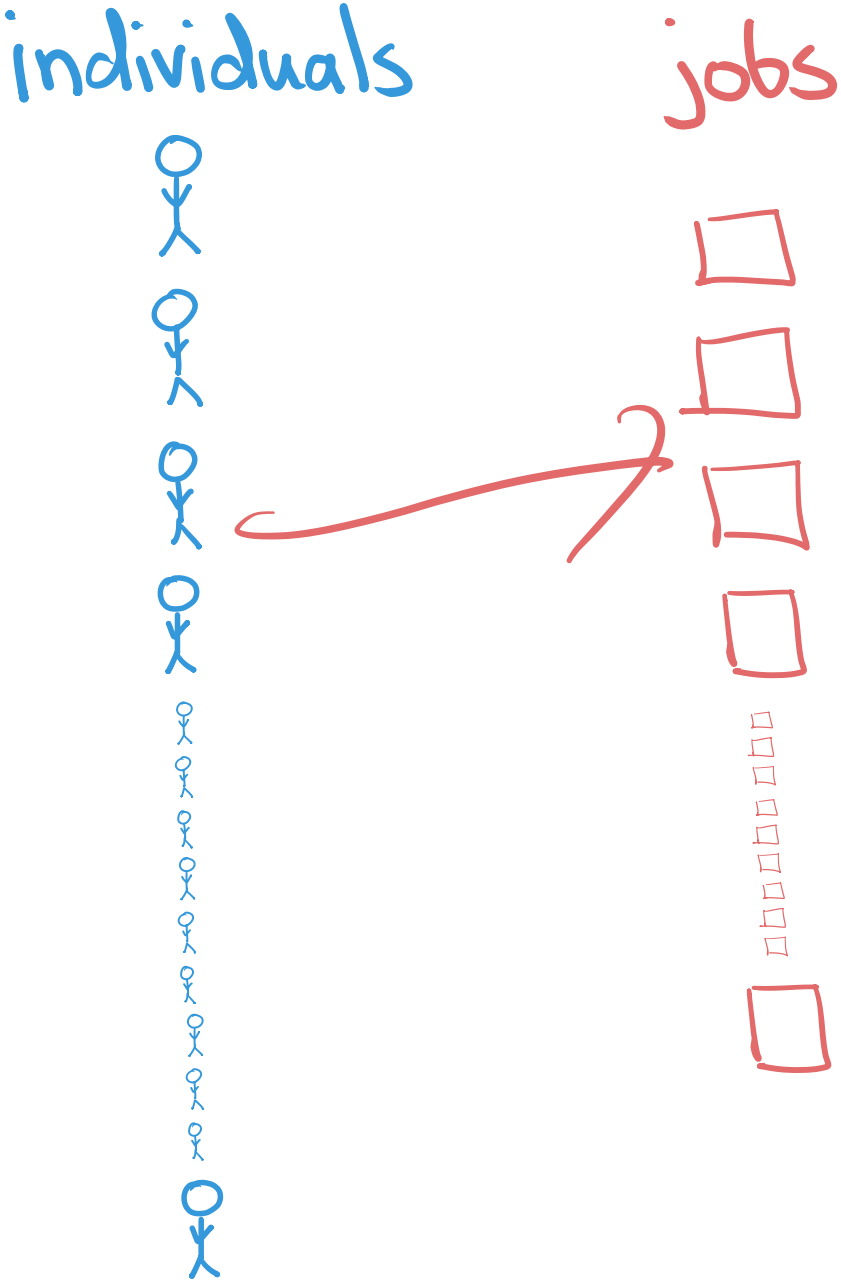
"Qualification matrix"

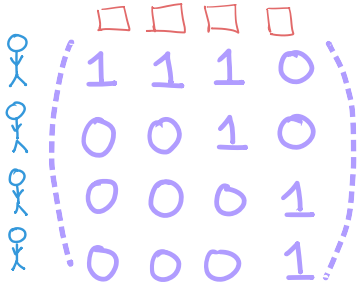


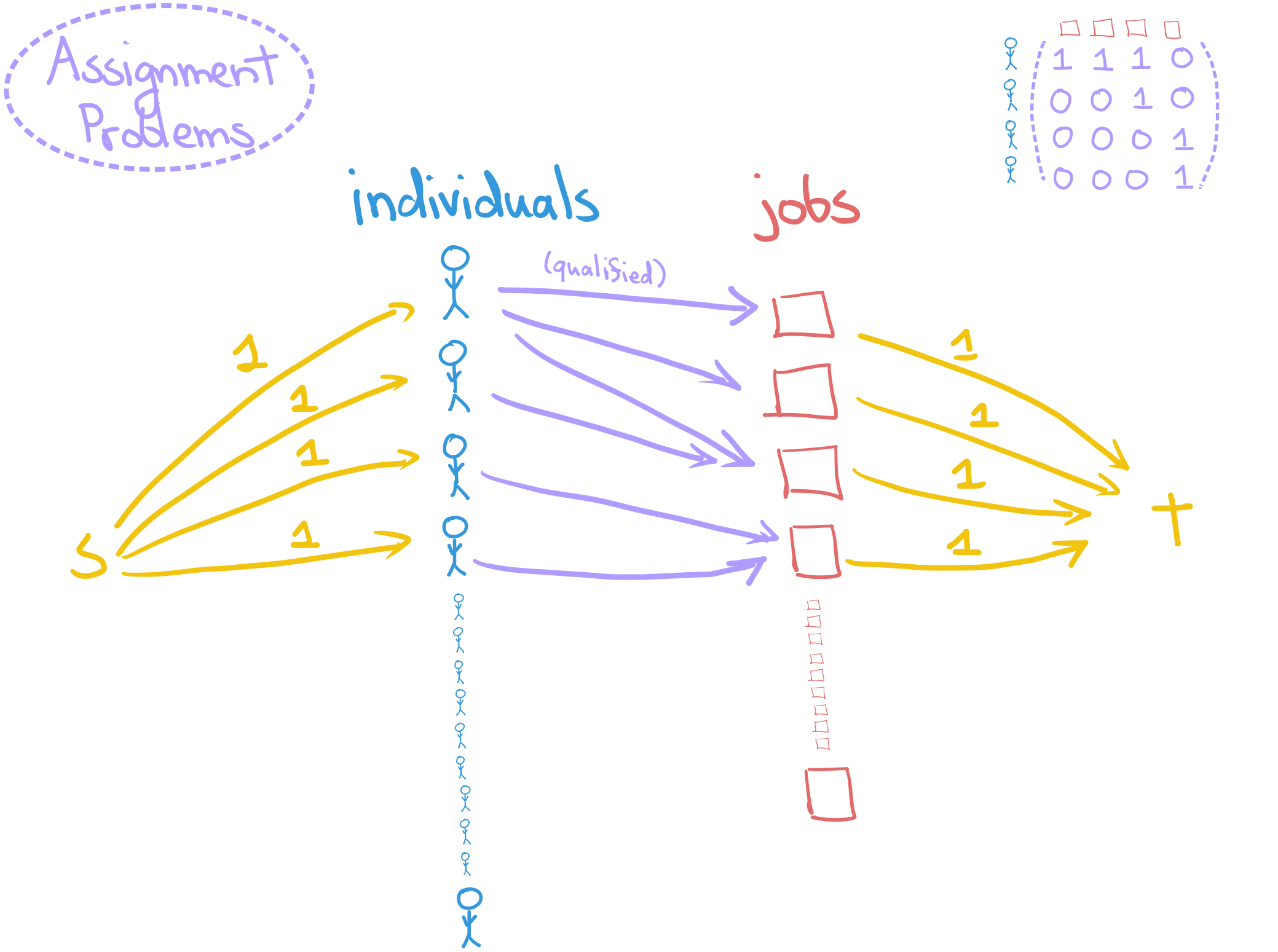
"1" = qualified

"0" = not qualified

Assignment Problems

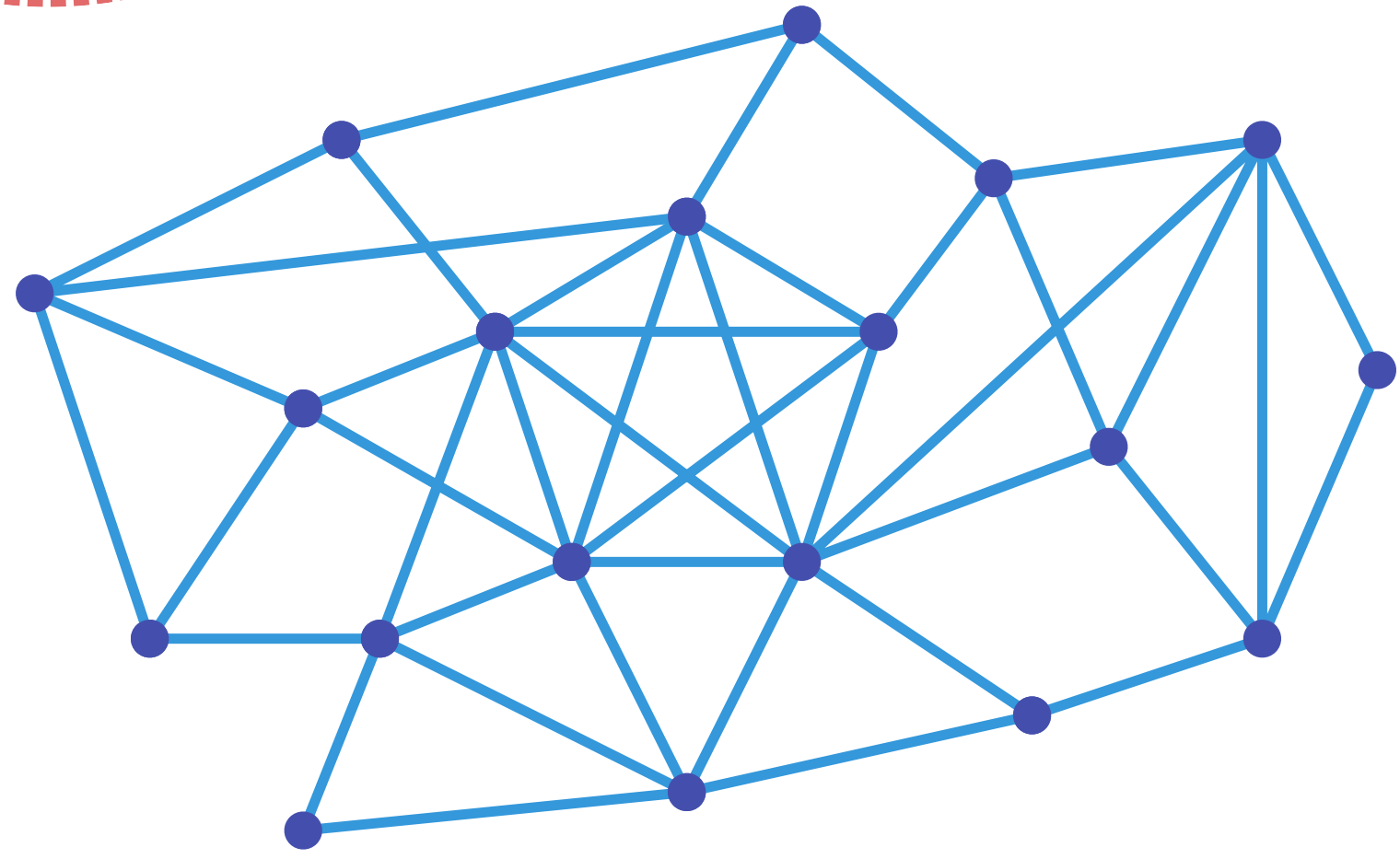





Assignment Problems
1 1 1 0
0 0 1 0
0 0 0 1
0 0 0 1
individuals
jobs
(qualified)
1
1
1
1
1
1
1
1
s
t

Edge Orientation

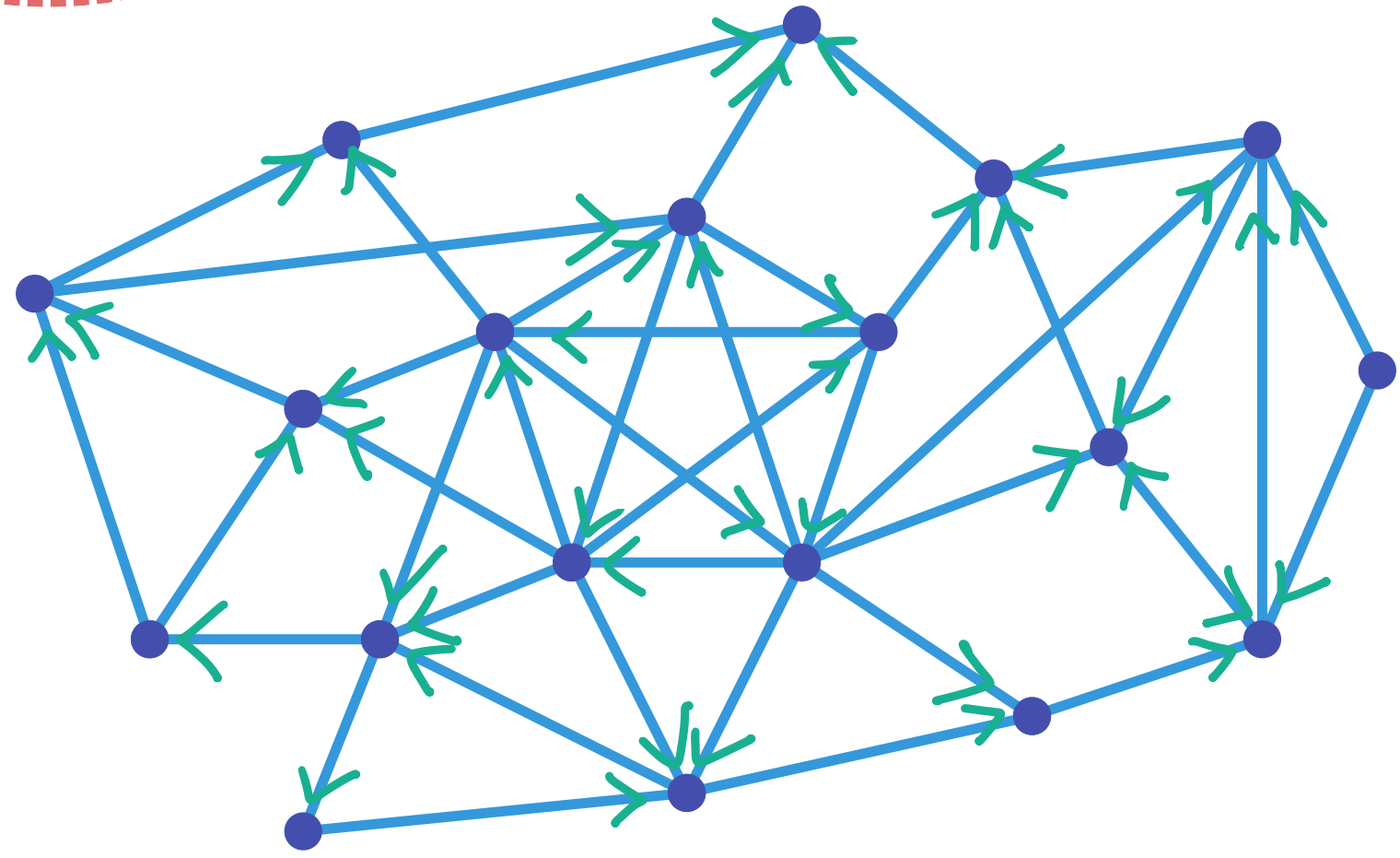
input: undirected graph



Goal: direct the edges to minimize the max density

Edge Orientation

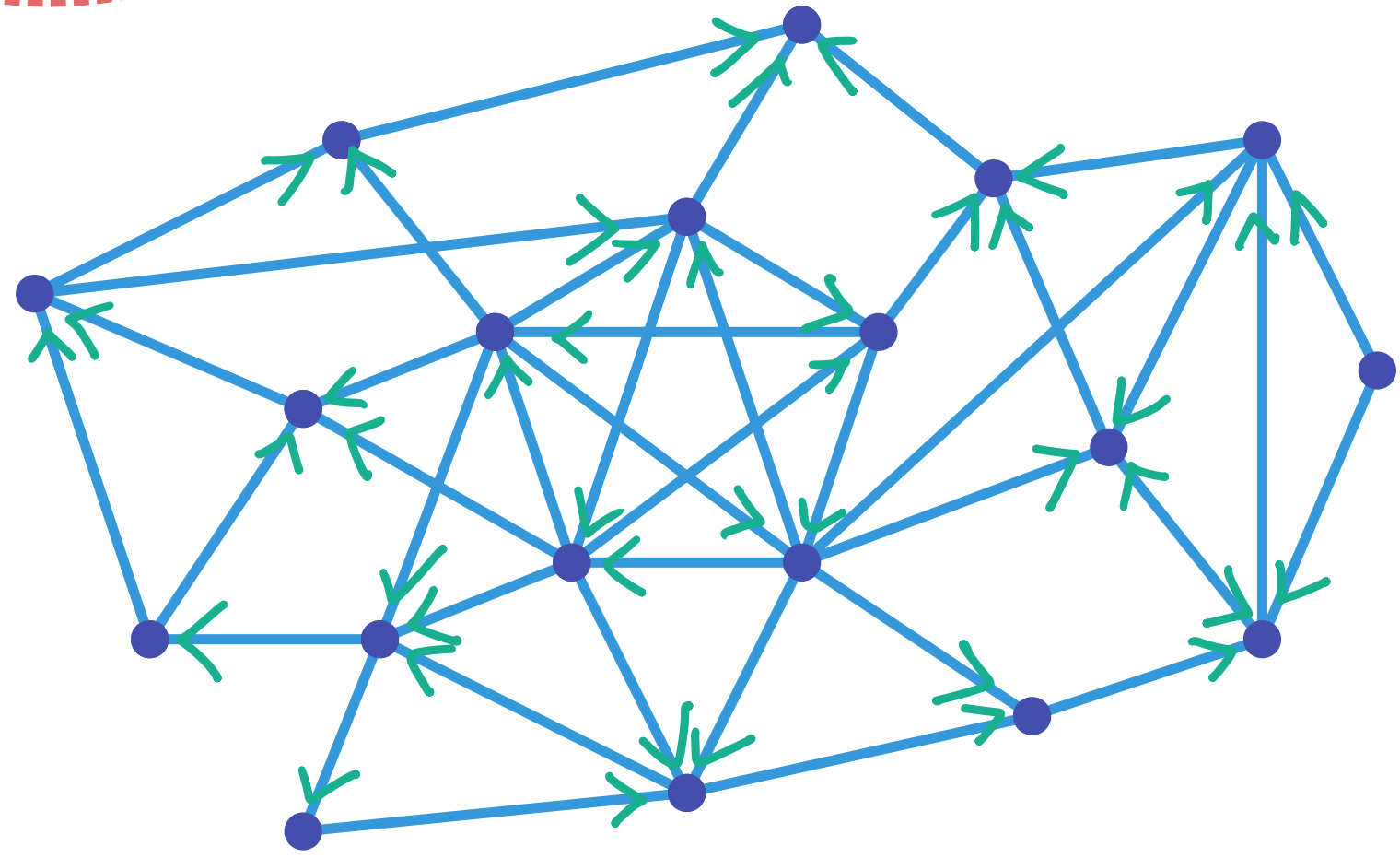
input: undirected graph



Goal: direct the edges to minimize the max in-degree

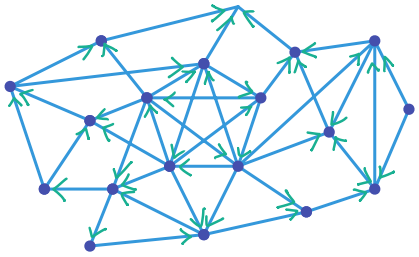
Edge Orientation

input: undirected graph



Decision version: given $h>0$, is there an orientation w/ max in-degree $\leq h$?

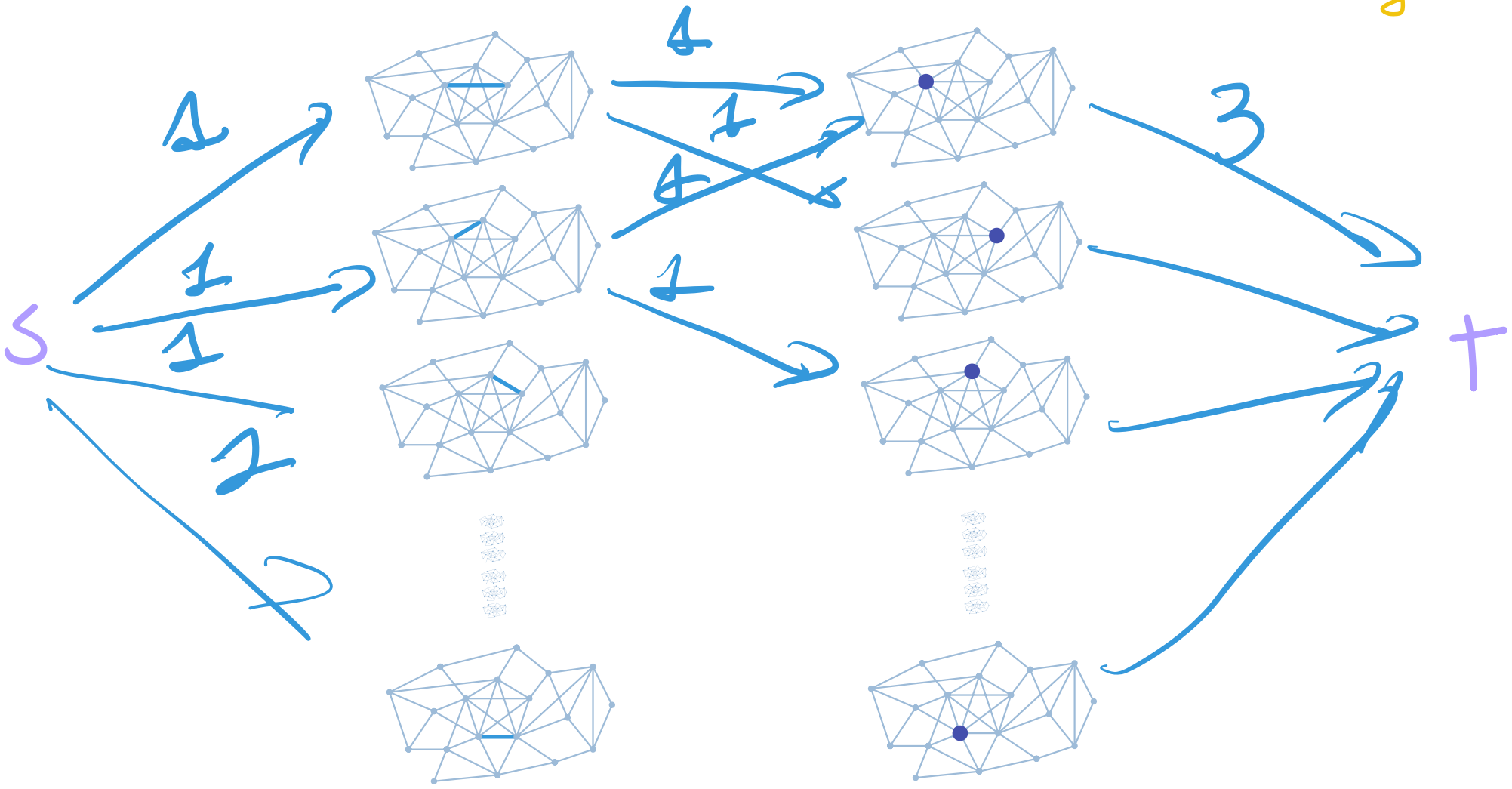
Edge Orientation



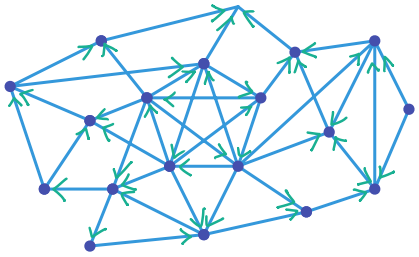
edges

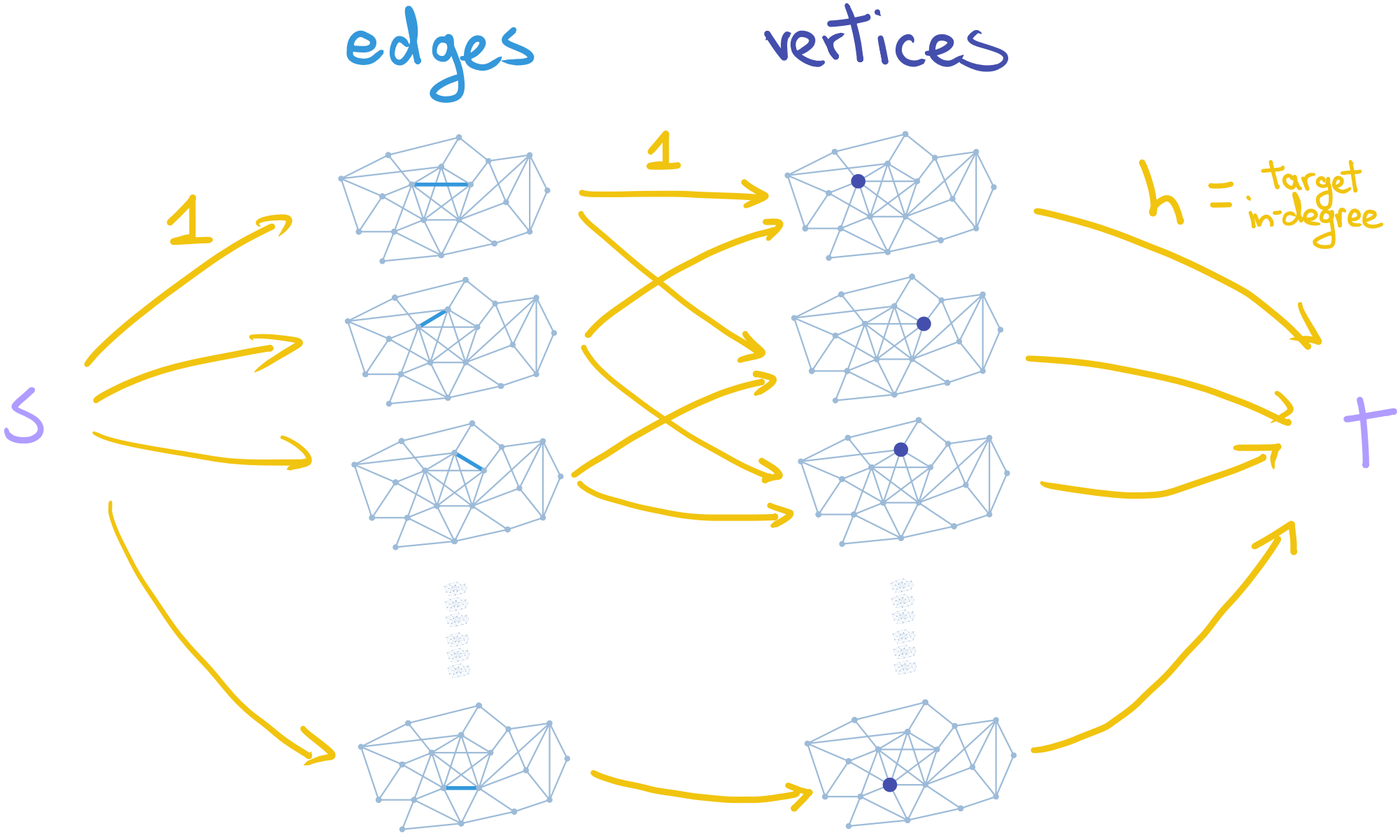
vertices

h = target in-degree



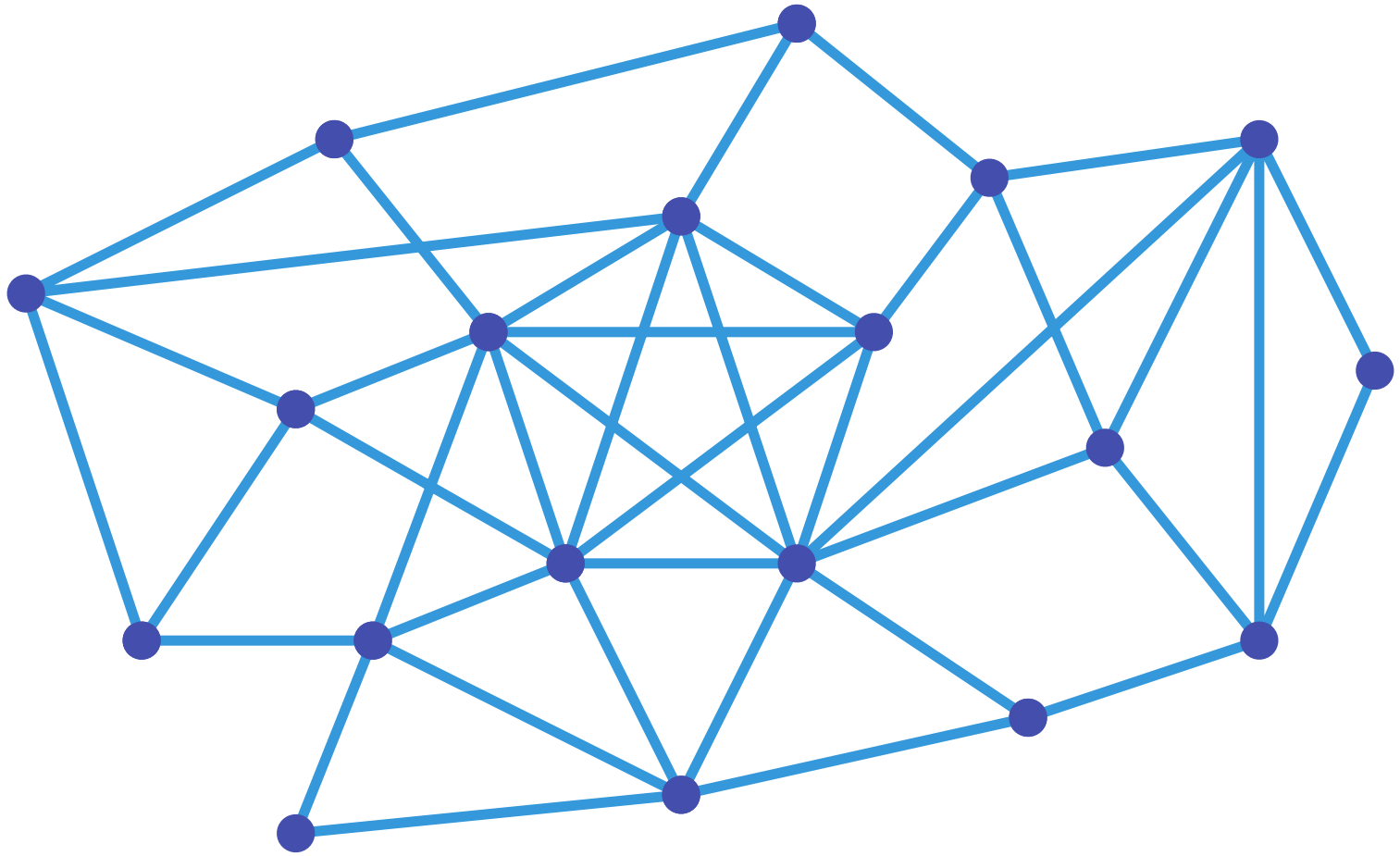
Edge Orientation





Densest Subgraph

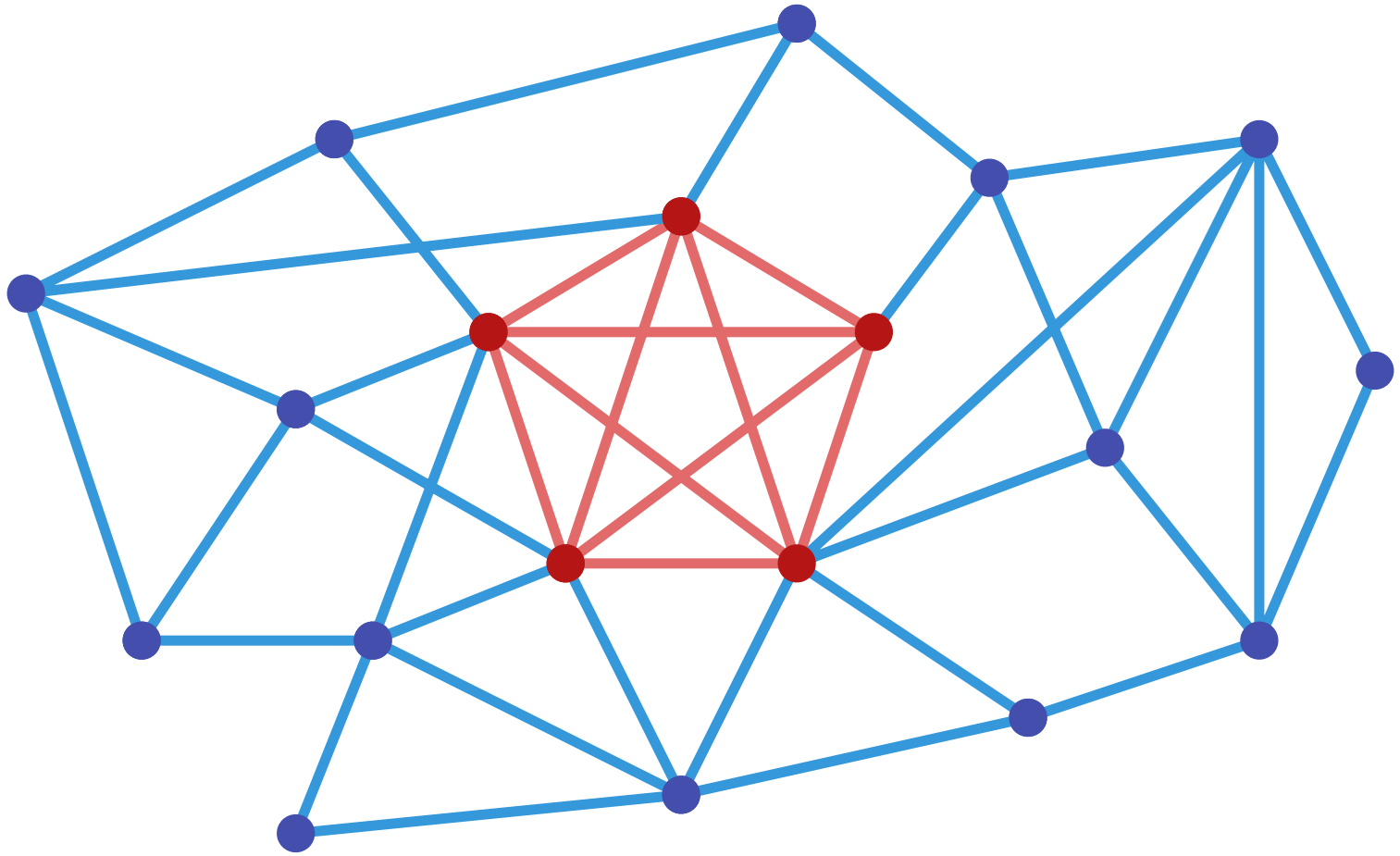
"density" = $\frac{\#\text{ edges}}{\#\text{ vertices}}$



Goal: compute densest subgraph

Densest Subgraph

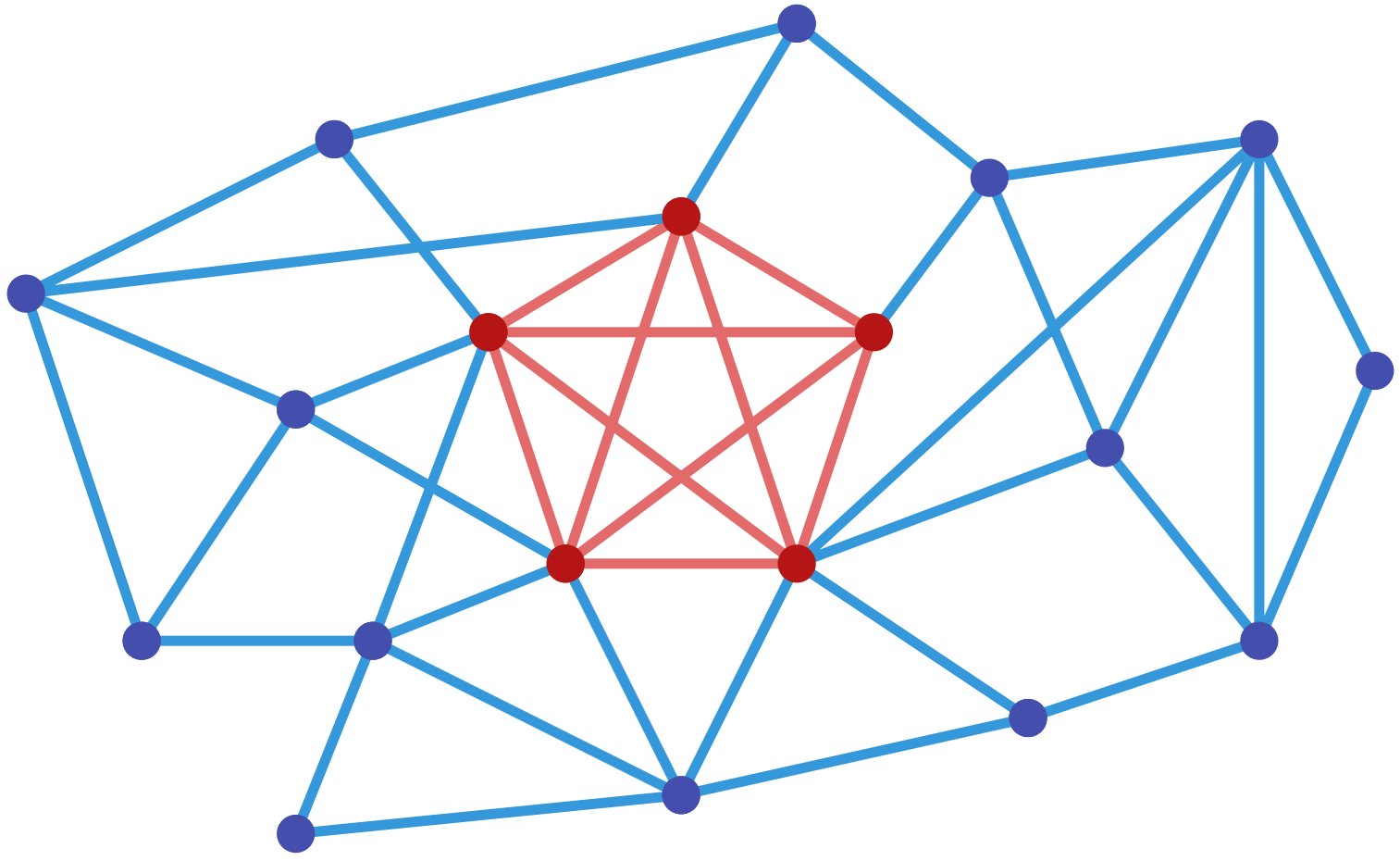
"density" = $\frac{\#\text{ edges}}{\#\text{ vertices}}$



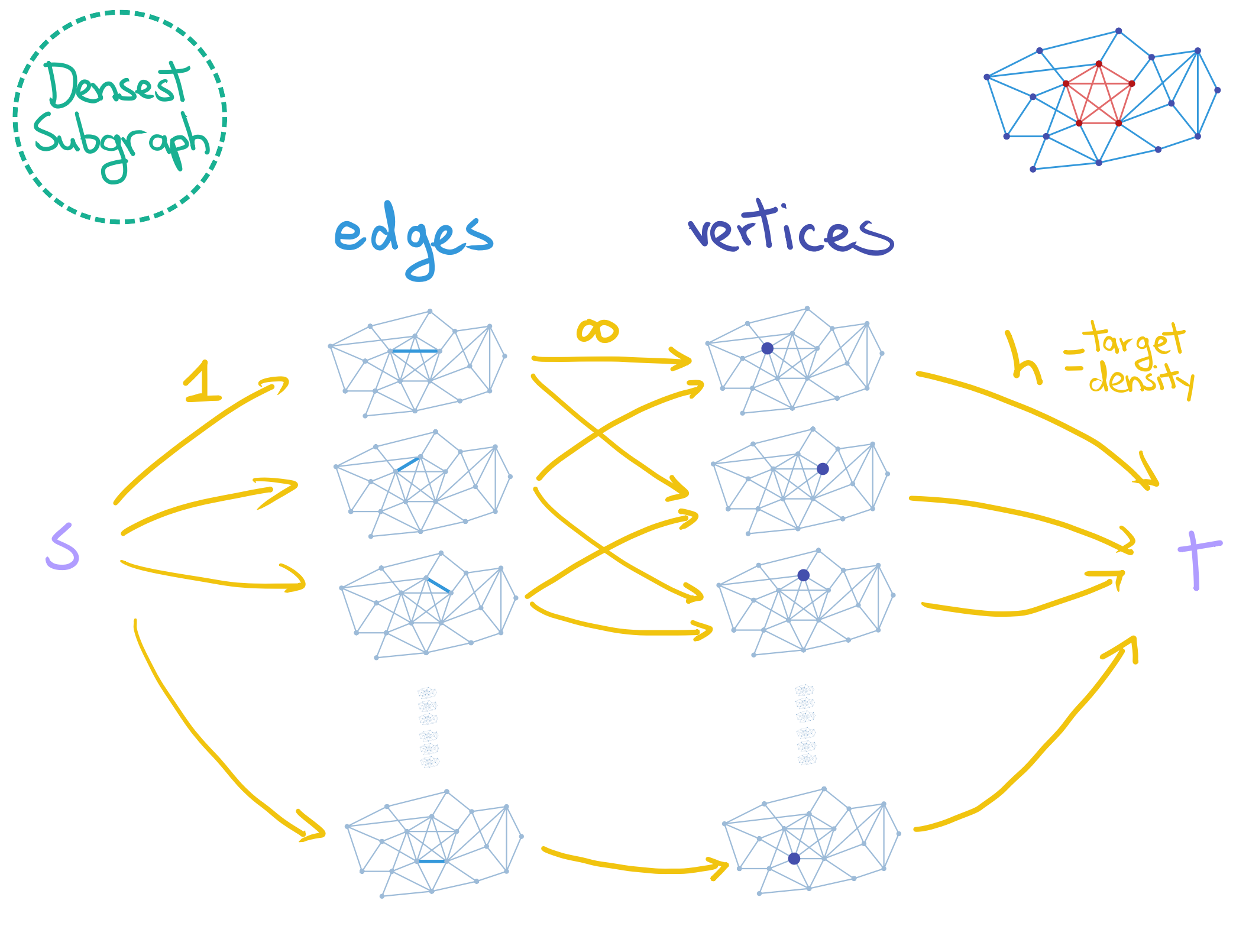
Goal: compute densest subgraph

Densest Subgraph

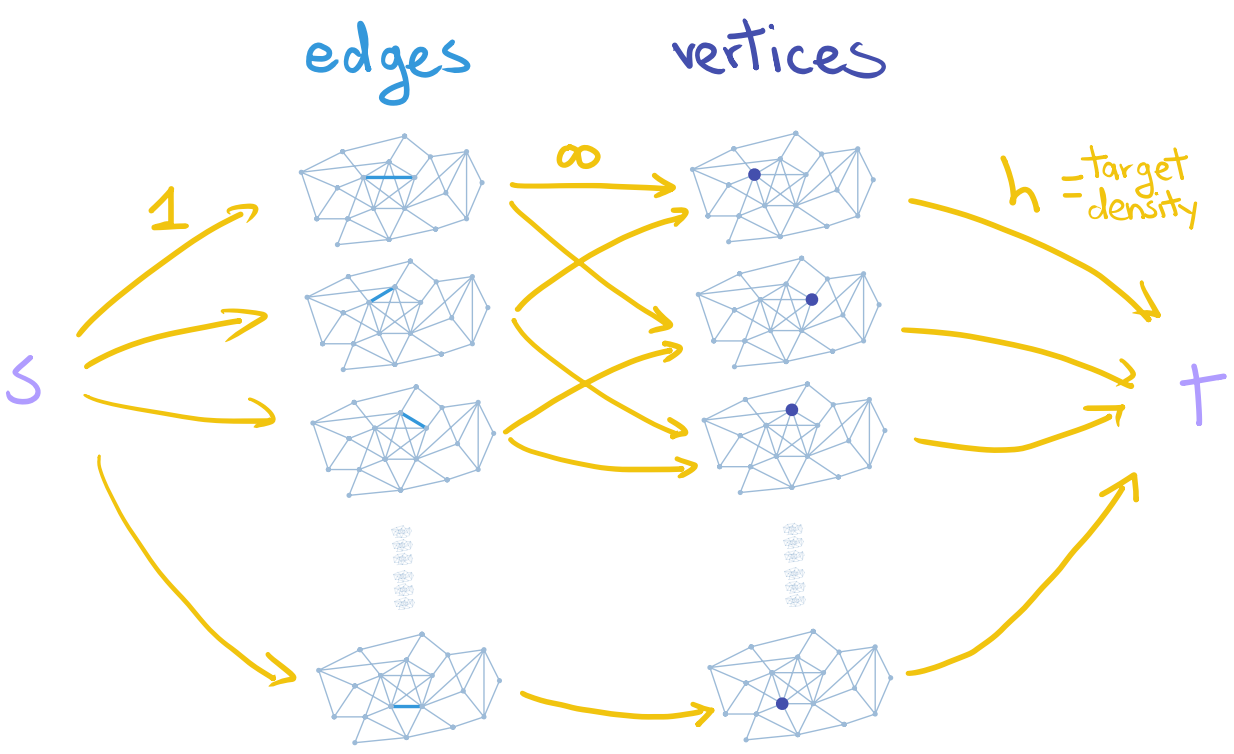
"density" = $\frac{\#\text{ edges}}{\#\text{ vertices}}$

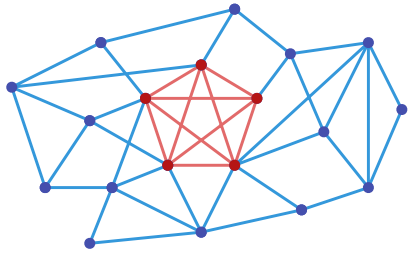


Decision version: Given $h>0$, is there a subgraph w/ density $>h$?


Densest Subgraph
edges
vertices
1
∞
h = target density
s
t

Densest Subgraph

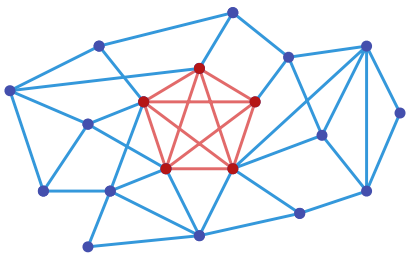


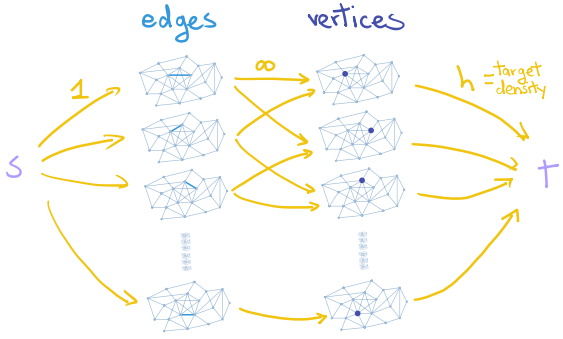


claim: if max-flow = m, then all subgraphs have density ≤ h

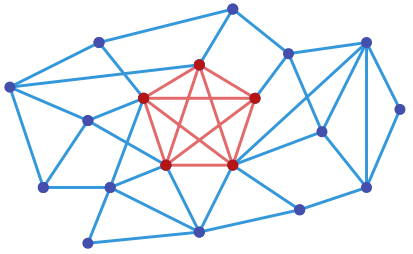
A subgraph of density > λ induces (s,t)-cut of size < m

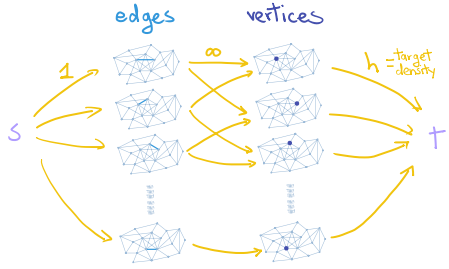
claim: if max-flow = m, then all subgraphs have density ≤ h



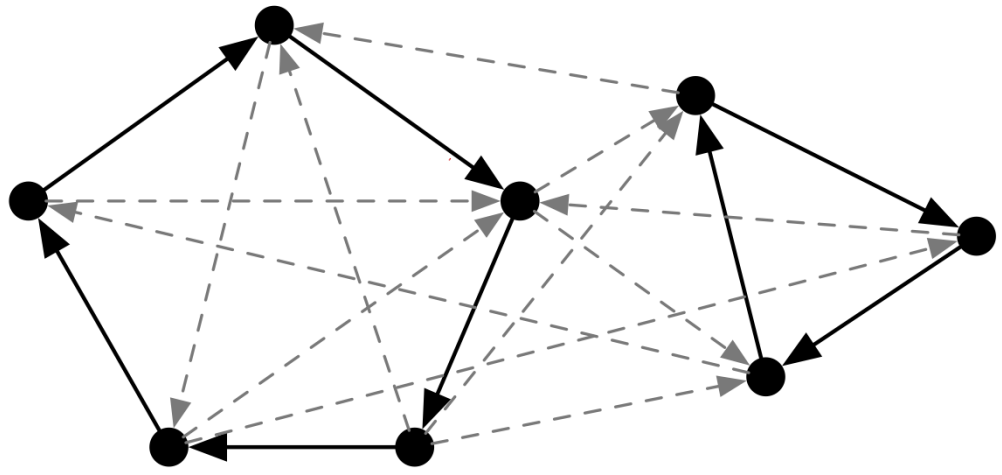


A subgraph of density $>\lambda$ induces (s,t)-cut of size $<m$





Exercise 19.10. A *cycle cover* of a given directed graph $G = (V, E)$ is a set of vertex-disjoint cycles that cover every vertex in G. You can also think of a cycle cover as a subset of edges $F \subseteq E$ such that each vertex v has exactly one edge in F entering v, and exactly one edge in F leaving v. This implies that F has exactly n edges. Below is a directed graph where the solid arcs form a cycle cover, and the dashed arcs are the remaining edges in G.



Consider the problem of deciding if a directed graph G contains a cycle cover. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

Exercise 19.3. Suppose we are given two-dimensional array $A[1..m][1..n]$ of non-negative real values such that each row and column sum is an integer. We want to round A to an integer matrix, replacing each entry $x \in A$ with either $\lceil x \rceil$ or $\lfloor x \rfloor$, while maintaining the sum in any row or column of A. For example,

$$\begin{pmatrix} 1.2 & 3.4 & 2.4 \\ 3.9 & 4.0 & 2.1 \\ 7.9 & 1.6 & 0.5 \end{pmatrix} \text{ rounds to } \begin{pmatrix} 1 & 4 & 2 \\ 4 & 4 & 2 \\ 8 & 1 & 1 \end{pmatrix}.$$

Consider the problem of computing such a rounding, or declaring that none exists. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

Exercise 19.8. Recall that in chess, a *rook* (♖) is a piece that can move horizontally or vertically as far as it wants. (It cannot move diagonally.) We say that a rook on a particular square is *attacking* another chess piece at another square if the rook can be moved horizontally or vertically from its square to the square of that chess piece.

Consider the following problem which we call the "n-rooks problem". At a high level, there is an $n \times n$ board where some of the squares have been marked off as "unavailable". The remaining squares are "available". The goal is to place n rooks in the available squares so that none of the rooks are attacking each other. We call this a "non-attacking placement" of the n rooks. (While rooks cannot be placed on unavailable squares, they can still move over unavailable squares to attack other pieces on the other side of unavailable squares.)

Below we draw a non-attacking placement of 4 rooks on a 4×4 chess board. Unavailable squares are marked by X. Rooks are marked by ♖.

X	♖		X
♖	X	X	X
	X	♖	
X		X	♖

Formally, the input consists of an $n \times n$ bit-array $A[1..n, 1..n] \in \{\texttt{true}, \texttt{false}\}^{n \times n}$, where $A[i, j] = \texttt{true}$ indicates that square (i, j) is available, and $A[i, j] = \texttt{false}$ indicates that square (i, j) is unavailable. The goal is to decide if there is a non-attacking placement of n rooks in the available squares. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

Exercise 19.9. This problem models the following scenario. To prepare for the final, you and your classmate have decided to try to solve all n of the exercises in the textbook. Due to time constraints, you've decided to split up the problems. To try to make the total effort even, each of you went through and skimmed all the problems, and for each problem, estimated the number of minutes it would take to solve the problem. (Thus producing two lists of estimates, one for each person.) The goal is to assign each exercise to you or our classmate as to minimize the maximum time that either of you would spend solving their assigned problems. We call this problem the "optimum study allocation problem".

Formally, we let $\{1, \ldots, n\}$ represent the n exercises. The input consists of two lists of natural numbers $A[1..n]$ and $B[1..n]$. For $i = 1, \ldots, n$, $A[i]$ represents your estimated time for problem i and $B[i]$ represents your classmate's estimated time for problem i. The goal is to divide $\{1, \ldots, n\}$ into two sets, $S \subseteq \{1, \ldots, n\}$ and $T = \{1, \ldots, n\} \setminus S$, to minimize the maximum of

$$\sum_{i \in S} A[i] \text{ and } \sum_{j \in T} B[j].$$

Here $\sum_{i \in S} A[i]$ represents the total time you estimate to solve all the problems in S and $\sum_{j \in T} B[j]$ represents the total estimated time for your partner to solve all the problems in T.

For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

Chapter 19

Applications of flows and cuts

There is a rich variety of problems that can be solved by maximum flow, edge-disjoint paths, and minimum cut. We survey a few of these problems and their reductions here. To encourage the reader to attempt to find the reductions themselves, this chapter is organized as follows. We first introduce each problem. Then we revisit each of these problems and show how solve them by reductions to flows and cuts.

19.1 The problems

19.1.1 Vertex disjoint paths, vertex capacitated flow, and vertex cuts

Thus far we have discussed edge-disjoint paths and edge-capacitated flows. However one can also model flow problems where there are constraints on the vertices. For example, it is natural to ask for the maximum number of *vertex-disjoint* (s,t)-paths. In the dual, a *vertex* (s,t)*-cut* refers to a set of vertices whose removal disconnects s from t. Here we assume that s and t are not connected by an edge.

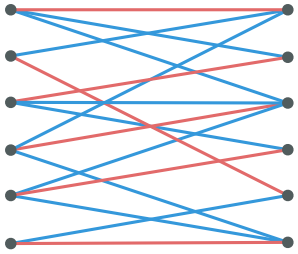
For example, in the following graph, there are two edge-disjoint paths from s to

t, and the minimum edge-cut is 2. But there is only one vertex disjoint path, and a vertex cut of size one (circled in the middle).

One can also consider combinations of vertex and edge capacities. This is a reasonable constraint in communication networks, where each node represents a switch or router. The edge capacities reflects how much data can be carried through each cable. The vertex capacities model how much data can be passed through each router.

19.1.2 Bipartite matching

Let $G = (V, E)$ be a bipartite graph. That is, suppose there is a partition of V into two sets L, R (mnemonics for "left" and "right"), such that every edge $e \in E$ has exactly one endpoint in L and one endpoint in R.



A *matching* is a set of edges $M \subseteq E$ where no two edges share an endpoint. The *bipartite matching* problem is to compute the maximum cardinality matching.

19.1.3 Project selection

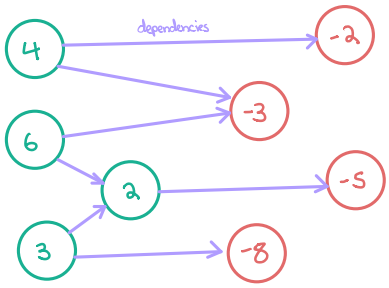
Here we consider a generic planning problem called *project selection*. The high level idea is that there are many possible projects that we can undertake. Some bring profit and others are net loss. What makes this problem interesting is that they depend on each other in complicated ways. Perhaps project A has to be completed before project B, and projects A and B both have to completed before project C. The goal is to choose a maximum profit subset of projects, where for each selected project, we have to also include the other projects that it depends on. Formally the input is as follows.

1. A set of projects, P.

2. For each project $p \in P$, a value $\mu(p) \in \mathbb{R}$ that measures the net profit of the project.

3. For each project $p \in P$, a set of *precedence constraints* $R(p) \subset P$, which reflects the set of projects that also have to be undertaken before p.

The following is a visual example where each vertex corresponds to a project. Each edge reflects a dependency – an edge (x, y) indicates that project x depends on project y.



We assume that the precendence constraints $R(p)$ induce a directed graph that is acyclic. (A directed cycle would give a list of projects $x_1, \ldots, x_k$ where each x_i has to be completed before x_{i+1} for each i, but also x_k has to be completed before x_k – which describes a deadlock from a project planning point of view.) A variant of the problem that allows for cycles, so to speak, is given in exercise 19.4.)

19.1.4 Playoff Elimination

Imagine we are nearing the end of a baseball season, and several teams are competing for one spot in the playoffs. We want to know if our favorite team has a chance of making the playoffs. Suppose we want to know if the Boston Red Sox have been eliminated from the playoffs, and that standings are as follows.

Team	Wins	Games left	
New York	92	2	
Baltimore	91	3	
Toronto	91	3	
Boston	90	+ 2	92

Since Boston has 2 games, it is possible for them to win both and catch up to the Yankees. But we still have to ensure that the Yankees don't win, and that neither Baltimore nor Toronto win more than 1 game each. That can get tricky if these teams are playing with each other. For example, suppose the remaining schedule specifies the following games between the following four teams:

(wins)	(92) NewYork	(91) Baltimore	(91) Toronto	(90) Boston
NewYork	×	1	1	0
Baltimore	1	×	1	1
Toronto	1	1	×	1
Boston	0	1	1	×

A close examination reveals it is impossible for Boston to catch up to first place: assuming Boston wins all their games, and the Yankees lose all their games, (which are both obviously necessary), the winner of the game between Baltimore and Toronto will end up with 93 wins.

Can we automate such an analysis? In general, the input is as follows.

1. A set of teams T.
2. For every team $t \in T$, the number of wins $w(t)$ that they already have.
3. For every pair of teams $s, t \in T$, the number of games $g(s,t)$ between them.
4. The home team $h \in T$.

The goal is to figure out if there's any set of outcomes to the remaining games so that the home team h is at least tied for the most wins.

19.1.5 Assignment

Bipartite matching and generalizations of bipartite matching help model *assignment problems*. Consider for example the following example from Kuhn [Kuh55]. We have m individuals and n jobs; the goal is to assign individuals to all the jobs. However not all individuals are qualified for all jobs. What we to encode these qualifications is via an $\{0,1\}^{m \times n}$ *qualification matrix*. Below we reproduce a 4×4 example from [Kuh55].

$$Q = \begin{pmatrix} 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

The qualification matrix states that individual 1 is qualified for jobs $1, 2, 3$; individual 2 is qualified for job 3, and individuals 3 and 4 are both qualified for jobs 4. The assignment problem asks, to quote [Kuh55]:

> *What is the largeset number of jobs that can be assigned to qualified individuals (with not more than one job assigned to each individual)?*

More general assignment problems. A more general version of the assignment job might, for example, state that certain jobs require a certain number of slots to be filled. Suppose that in addition to the qualification matrix above, we also have two sets of cardinalities:

1. $a_i \in \mathbb{N}$ for each individual $i \in [m]$, which says that individual i can take on at most a_i jobs.
2. $b_j \in \mathbb{N}$ for each job $j \in [n]$, which says that we have b_j slots of job j to try to fill.

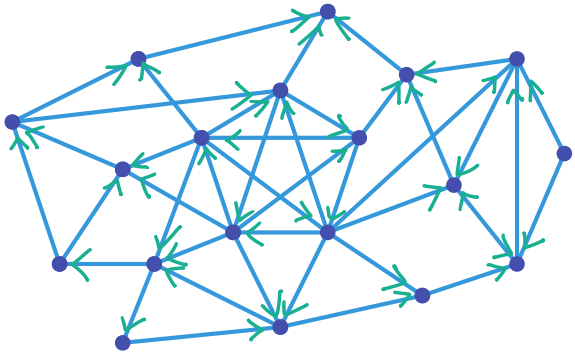
A particular individual can only be assigned to a particular job once.

The "simple" assignment version above was the case where $a_i = b_j = 1$ for all i and j.

This more general version of the assignment problem asks:

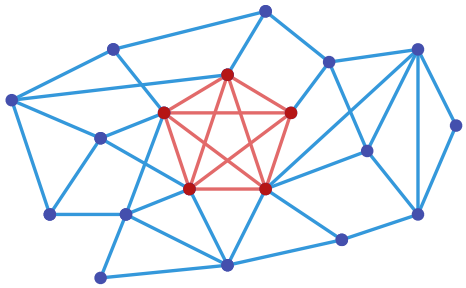
> *What is the largest number of jobs (counted with multiplicity) that can be assigned to qualified individuals?*

19.1.6 Edge orientation



Let $G = (V, E)$ be an undirected graph. The *edge orientation* problem is to assign directions to every edge $e \in E$ while minimizing the maximum in-degree in the resulting directed graph. There is also a parametrized version of the problem where one is given a parameter $h \in \mathbb{N}$, and the goal is to compute an orientation where the maximum degree is at most h (or declare that this is impossible).

19.1.7 Densest subgraph



Let $G = (V, E)$ be an undirected graph. The *density* of a graph G is defined as the ratio

$$(\text{density of } G) = \frac{|E|}{|V|}.$$

The *densest subgraph problem* is to compute the subgraph of G with greatest density. Equivalently, we want to select a set of vertices S maximizing the ratio $|E_S|/|S|$, where we let $E_S = \{e \in E : e \subseteq S\}$ denotes the set of edges with both endpoints in S.

Among other applications, densest subgraphs arises in the analysis of social networks, where a very dense subgraph can be interpreted as a tight-knit community.

19.2 Reductions to flows and cuts

We now explain how to solve each of the problems introduced above.

19.2.1 Vertex disjoint paths and vertex capacities

Recall the problem of vertex-capacitated flow from section 19.1.1.

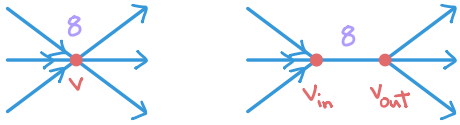
Theorem 19.1. *Let $G = (V, E)$ be a directed graph with positive vertex capacities $c : V \to \mathbb{R}_{>0}$. Let $s, t \in V$ be a fixed source and sink, respectively, and suppose there is no edge from s to t.*

1. *The maximum vertex capacitated (s, t)-flow equals the minimum capacity (s, t)-vertex-cut.*

2. *If the capacities are integral, then the maximum vertex capacitated flow (s, t)-flow is attained by an integral flow.*

3. *The maximum vertex capacitated (s,t)-flow and the minimum capacity (s,t)-vertex-cut can be reduce to edge-capacitated flows and cuts in a graph with $O(m+n)$ edges and $O(n)$ vertices.*

Proof sketch. We reduce from vertex capacities to edge capacities with the following simple trick. For each vertex v with capacity $c(v)$, we split v into two vertices, "v-in" v^- and "v-out" v^+. All edges directed into v are instead directed into v^-, and all edges directed out of v are instead directed out of v^+. We add an edge from v^- to v^+ with capacity $c(v)$.



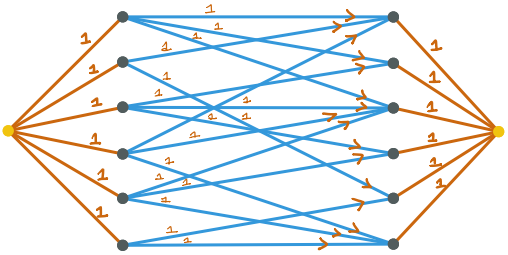
Meanwhile, for the "normal" edges, we assign each of them capacity $+\infty$, since there is no limit to the amount of flow on that edge.[1] We leave it to the reader to apply the ideas from edge capacitated flow to this auxiliary graph to give the claims. □

19.2.2 Bipartite matching

Recall the bipartite matching problem from section 19.1.2. We are given an undirected bipartite graph G and the goal is to compute the maximum cardinality matching.

Theorem 19.2. *The maximum cardinality matching reduces to an edge-disjoint paths problem with $O(m+n)$ edges and $O(n)$ vertices.*

Proof. The high-level ideas are sketched below.



Formally, suppose we are given a bipartite graph $G=(V,E)$, letting $L,R \subseteq V$ denote the two parts. We introduce a super-source vertex $s^\star$ on the left and a super-sink $t^\star$ on the right. We add an edge with capacity 1 from $s^\star$ to every $\ell \in L$ and an

[1] This is particularly convenient for computing the minimum vertex (s,t)-cut, since the $+\infty$-capacity edges are always omitted.

edge with capacity 1 from every $r \in R$ to $t^\star$. We also direct all the edges in E from L to R, and give each edge capacity 1.

Every path from $s^\star$ to $t^\star$ is of the form $(s^\star, \ell, r, t^\star)$, where $\ell \in L$, $r \in R$, and $\{\ell, r\} \in E$. For each edge $e = \{\ell, r\}$, let

$$p_e \stackrel{\text{def}}{=} (s^\star, \ell, r, t^\star)$$

denote that $(s^\star, t^\star)$-path via e. The mapping from $e \in E$ to p_e defines a bijection between E and $(s^\star, t^\star)$.

We claim that a set of edges $M \subseteq E$ is a matching iff the corresponding paths $\{p_e : e \in M\}$ is edge-disjoint. If so, then a maximum packing of edge-disjoint paths gives a maximum matching.

Let M be a matching. Since the endpoints of M are distinct, each vertex $\ell \in L$ appears in at most edge in M. Consequently the edge $(s^\star, \ell)$ appears in at most one path. Likewise for each $r \in R$, the edge $(r, t^\star)$ appears in at most one path. Lastly, we observe that each of the arcs (ℓ, r) appears in at most one path (namely, $p_{\{\ell,r\}}$). We conclude that the paths are edge-disjoint.

Conversely, suppose $\{p_e : e \in M\}$ is edge-disjoint. For each vertex $\ell \in L$, since $(s^\star, \ell)$ appears in at most one path p_e, ℓ is an endpoint to at most one edge in M. Similarly, for each vertex $r \in R$, since $(r, t^\star)$ appears in at most one path p_e, r is an endpoint to at most one edge in M. Thus M has distinct endpoints, and is a matching. □

19.2.3 Assignment

Recall the (simple) assignment problem from section 19.1.5. We reduce the problem to edge-disjoint paths and the construction is essentially the same as bipartite matching.

1. We create a vertex ℓ_i for each individual $i \in [m]$, and a vertex r_j for each job $j \in [n]$.
2. We add a directed edge (ℓ_i, r_j) whenever individual i is qualified for job j.
3. We add a super-source vertex s, and a directed edge from s to ℓ_i for each $i \in [m]$.
4. We add a super-sink vertex t, and a directed edge from r_i to t for each job $j \in [n]$.

By construction, all (s, t)-paths are of the form (s, ℓ_i, r_j, t) for $i \in [m]$ and $j \in [n]$ such that individual i is qualified for job j. That is, we have a bijection between (s, t)-paths and pairs (i, j) where i is qualified for job j.

Now consider a family of edge-disjoint (s,t)-paths. For each $i \in [m]$, there is at most one path containing ℓ_i because ℓ_i has only one incoming edge. For each $j \in [n]$, So the individual-to-job assignments corresponding to a family of edge-disjoint paths will feature each individual and each job at most once.

Conversely suppose we have an assignment where each individual and each job is assigned at most once. We claim the corresponding family of (s,t)-paths will be edge-disjoint. Indeed:

1. Each edge of the form (s, ℓ_i) is used at most once, because individual i is assigned at most once.

2. Each edge of the form (r_i, t) is used at most once, because each individual j is assigned at most once.

3. Each edge of the form (ℓ_i, r_j) is used at most once because individual i is assigned at most once (or also because job j is assigned at most once).

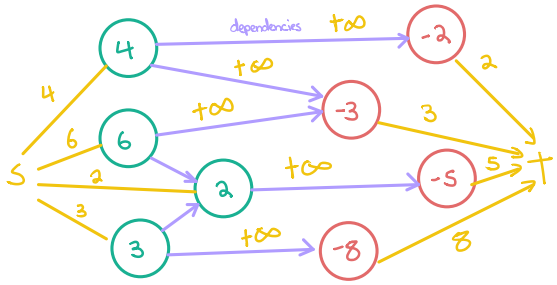
Thus there is a feasible assignment of size k iff there are k edge-disjoint (s,t)-paths, and we can map one family to the other as described above. We run the augmenting paths algorithm to get a maximum family of (s,t)-edge disjoint paths, and extract the assignment as described above. This takes $O(mn \min\{m,n\})$-time because there are at most $O(mn)$ edges, and the maximum size of the flow (so to speak) is $\min\{m,n\}$.

19.2.4 Project selection

Recall the project selection problem from section 19.1.3.

Theorem 19.3. *The project selection problem with n projects and m dependencies reduces to computing the (s,t)-min cut in a flow network with $O(m+n)$ edges and $O(n)$ vertices.*

Proof sketch. We reduce to (s,t)-min-cut in the following flow network. We introduce a source s and add a directed edge from s to every profitable project. The edge is given capacity equal to the profit. We introduce a sink t, and a directed edge from every negative profit project to t. Each such edge is given capacity equal to the (positive) cost of that project. We add a directed edge for every dependency with capacity $+\infty$.



Consider any set $S \subset V$ that contains s and excludes t, and the induced (s,t)-cut $\delta^+(S)$. If the capacity of the cut is finite, then for every project p in S, every dependency of p must also be in S. That is, S is a feasible project selection. Note that the minimum (s,t)-cut has finite capacity, since (for instance) $\delta^+(\{s\})$ has finite capacity.

Now suppose the capacity of the cut $\delta^+(S)$ is finite. All of the edges in the cut $\delta^+(S)$ are one of two types.

1. The edge from a source to a profitable project excluded from S.
2. The edge from a negative profit project in S to t.

Thus the capacity of the cut is exactly equal to

$$\left(\begin{array}{c}\text{sum of profits of positive}\\ \text{projects excluded from } S\end{array}\right) - \left(\begin{array}{c}\text{sum of costs of}\\ \text{negative projects in } S\end{array}\right).$$

We can rewrite this as

$$\left(\begin{array}{c}\text{total sum of profits}\\ \text{of all positive projects}\end{array}\right) - \left(\begin{array}{c}\text{sum of profits of all}\\ \text{positive projects in } S\end{array}\right) - \left(\begin{array}{c}\text{sum of costs of}\\ \text{negative projects in } S\end{array}\right)$$
$$= \left(\begin{array}{c}\text{total sum of profits of}\\ \text{all positive projects}\end{array}\right) - (\text{net profit of } S).$$

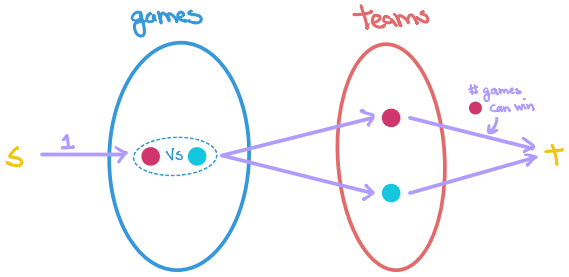
Note that the total sum of profits of all positive projects is fixed independently of S. Thus the minimum (s,t)-cut will maximize the net profit of the source side of the cut. □

19.2.5 Playoff Elimination

Recall the playoff elimination problem from section 19.1.4.

Theorem 19.4. *An instance of deciding playoff elimination with n teams, and m (unordered) pairs of teams with at least 1 game remaining, reduces to computing the maximum flow in a graph with $O(m+n)$ vertices and edges.*

Proof. We interpret playoff elimination as an allocation problem. We want to allocate one win from every game, while limiting the number of wins of all the non-home teams to keep the home team from being eliminated.



Let M be the maximum number of games the home team can win, if they win the rest of their games. We create a directed acyclic graph with 4 layers, as follows.

In the first layer we have a source s.

In the second layer we have a vertex $u_{\{a,b\}}$ for every pair of teams (a, b) with at least one game left to play. For each such pair (a, b), we add an edge from s to $u_{(a,b)}$ with capacity equal to the number of games left between teams a and b.

In the third layer we have a vertex v_a for every team a. For every team a, and every team b that a has left to play, we add an edge from the vertex $u_{\{a,b\}}$ to v_a with infinite capacity.

In the final layer we have a sink vertex t. For each team a except for the home team, we add an edge from v_a to t with capacity $M - w(a)$, where $w(a)$ is the number of wins team a already has. For the home team h, we add an edge from h to t with capacity $+\infty$.

Now, let G be the total number of games left to play. We compute the maximum flow in the graph described above. If the maximum flow has size G, then we declare that the home team has not yet been. Otherwise we declare that the home team has been eliminated.

Below, we draw the flow network described above for the example described in section 19.1.4.

We now prove that our reduction is sound. We first point out that the home team has not yet been eliminated iff there is a set of outcomes to all the games such that no other team has more than M wins. In one direction, suppose we have a set of outcomes where the home team makes the playoffs. We first reassign any game the home team loses so that the home team wins. The home team still makes the playoffs.

Now the home team has M wins, and has at least as many wins as any other team, so this gives an outcome where no other team has more than M wins.

Conversely, suppose there is a set of outcomes where no other team has more than M wins. We can again reassign any game the home team loses so that the home team wins. Now the home team has M wins, while every other team still has at most M wins, giving a set of outcomes where the home team makes the playoffs.

Now we will show that there is a max flow of size G iff there is a set of outcomes where no other team wins more than M games.

Suppose first that there is an outcome of the games where every other team finishes with at most M wins. Changing the outcome of the home team's games if necessary, we may assume that the home team finishes with M wins. From these outcomes we construct a flow of size G that saturates all the edges from s, and where each matchup-vertex $u_{(a,b)}$ sends one unit of flow to team-vertex v_a for every game between a and b won by a, and one unit of flow to team-vertex t_b for every game won by b. Each team-vertex v_a gets an amount of flow equal to the number of games won by a, which is at most $M - w(t)$, and we send to t from the (v_a, t)-edge. This gives a feasible flow of size G in our auxiliary network.

Conversely suppose there is an (s, t)-flow of size G. We will construct an outcome of games where every team has at most M wins. We may assume the flow is integral because the capacities are integral, and decompose the flow as an integral path-packing of G paths. Each path is of the form $(s, u_{a,b}, v_a, t)$ for a pair of teams a and b. We also observe that the total capacity of the edges leaving s is G, which forces each $(s, u_{a,b})$-edge to be saturated, hence the number of paths involving $v_{a,b}$ equals the number of games left between a and b.

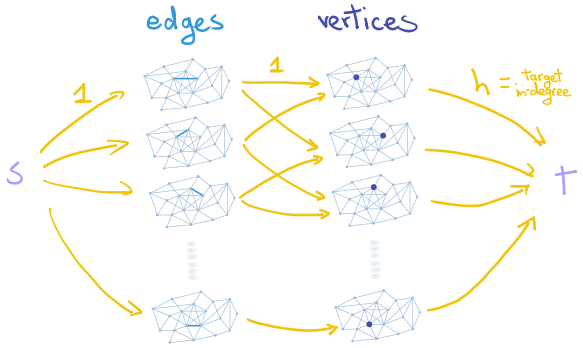
We create a set of outcomes where for each path $(s, u_{a,b}, v_a, t)$, we give one win from a matchup between a and b to a. This assigns an outcome to all G remaining games. Moreover, each team-vertex v_a can be in at most $M - w(a)$ paths since this is the capacity of the edge (v_a, t). Thus each team a ends up with at most M wins in this allocation. □

19.2.6 Edge orientation

Recall the edge orientation problem from section 19.1.6.

Theorem 19.5. *Let $G = (V, E)$ be an undirected graph with m edges and n vertices, and $h \in \mathbb{N}$, and consider the problem of deciding whether there is an orientation with maximum in-degre of h. This problem reduces to an edge-disjoint paths problem with $O(m)$ edges and $O(n)$ vertices.*

Proof. We can re-interpret edge orientation as an assignment problem with capacities. "Orienting an edge" is like assigning an edge to one of its endpoints. We want to see edges to endpoints so that each vertex is assigned at most $O(h)$ edges. We create a flow network similar to assignment.



We initially have a bipartite graph where one side we have an auxiliary vertex a_e corresponding to each edge $e \in E$, and on the other side we have an auxiliary vertex b_v corresponding to each vertex $v \in V$. We have an edge from a_e to b_v whenever v is an endpoint of e. We introduce an auxiliary source s and an auxiliary source t. We add an edge with capacity 1 from s go a_e for every e. We add an edge with capacity h from b_v to t for every vertex v.

After constructing this auxiliary graph, we compute a maximum path packing from s to t, and return `true` iff we obtain a path packing of size m.

To prove the reduction is sound, we show that there is a 1-1 correspondence between orientations with maximum in-degree h, and path-packings of size m. Given an orientation with maximum in-degree h, consider the path packing where for every edge e oriented toward an endpoint v, we take the path (s, a_e, b_v, t). This gives m paths that fits in the capacities; in particular, each edge of the form (b_v, t) is used at most h times because v has in-degree at most h in the orientation.

The inverse mapping takes path packings of size m and converts them into orientations with maximum in-degree h. We first observe that every path in a path packing is of the form (s, a_e, b_v, t), where v is an endpoint of v. Since there are m edges from s to distinct a_e's, in a path packing of m edges, there is exactly one path containing each a_e. For each e, we orient e towards the endpoint v where b_v is the next vertex in the corresponding path. This gives an orientation of every edge. Additionally, the maximum in-degree is at most h because that is the capacity of the edge (b_v, t).

This correspondance implies that there is an orientation with in-degree h iff there is a path packing of size m in the auxiliary graph, which completes the proof of

correctness. □

Corollary 19.6. *Consider the problem of computing the minimum in-degree orientation in a graph G with m edges and n vertices. Then this problem can be solved via $O(\log n)$ instances of edge-disjoint paths with $O(m)$ edges and $O(n)$ vertices.*

Proof. In theorem 19.5, we showed that the decision problem (given a target in-degree h) reduces to a single instance of edge-disjoint paths. We reduce the minimization problem (of finding the minimum feasible h) to $O(\log n)$ instances of the decision problem by binary search. Here observe that the optimum value of h is an integer between 1 and m. To "probe" a value of h, we apply theorem 19.5 to see if there exists an orientation with maximum in-degree h. If so, then the optimum values is $\leq h$. If not, then the optimum value is greater. We guide our binary search accordingly. □

19.3 Additional notes and references

We recommend [KT06] and [Eri19] for further discussions on applications of flow (including several of the applications discussed here).

Spring 2024 lecture notes. Click on the links below for the following files:
- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Spring 2023 lecture materials. Click on the links below for the following files:
- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Spring 2022 lecture materials. Click on the links below for the following files:
- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

19.4 Exercises

Exercise 19.1. Given integer values $b : \mathbb{R}^V$, a *b-matching* is a set of edges $M \subseteq E$ where every vertex v is incident to at most $b(v)$ edges in M. Consider the problem of computing the largest b-matching in a bipartite graph. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

Exercise 19.2. Every semester we have to schedule PSO's for all the students, which is complicated by (a) room availability mixed with (b) time availability both with respect to the students and the rooms. Suppose there are m total students, n available time slots, and p different rooms. Suppose we had the following information.

1. For each student (indexed by) $i \in [m]$, there is a subset of times $T_i \subseteq [n]$ of times when they can attend PSO.
2. For each time slot $j \in [n]$, there is a subset of rooms $R_j \subseteq [m]$ available at that time.
3. For each room $k \in [p]$ there is a maximum capacity C_k of the number of students that can fit in the room at any given time.

From this information we have the following scheduling problems.

1. Consider the problem of deciding if there is a way to schedule all the students to times and rooms while respecting all the constraints listed above. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.
2. In addition to making sure each room can fit all its students, we also want to minimize the maximum number of students in any room at any time, to decrease the maximum load on the TA's. Consider the problem of creating a schedule that minimizes the maximum number of students in any room, while also respecting the constraints listed above. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

Exercise 19.3. Suppose we are given two-dimensional array $A[1..m][1..n]$ of non-negative real values such that each row and column sum is an integer. We want to round A to an integer matrix, replacing each entry $x \in A$ with either $\lceil x \rceil$ or $\lfloor x \rfloor$, while maintaining the sum in any row or column of A. For example,

$$\begin{pmatrix} 1.2 & 3.4 & 2.4 \\ 3.9 & 4.0 & 2.1 \\ 7.9 & 1.6 & 0.5 \end{pmatrix} \text{ rounds to } \begin{pmatrix} 1 & 4 & 2 \\ 4 & 4 & 2 \\ 8 & 1 & 1 \end{pmatrix}.$$

Consider the problem of computing such a rounding, or declaring that none exists. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

Exercise 19.4. Recall the project selection problem in section 19.1.3. Suppose we relax the problem so that the "precedence requirements" are no longer required to be acyclic. We instead interpret each set $R(p)$ as "co-requisites": if we take project p, then we also have to take on project q for every $q \in R(p)$. (That is, we remove the emphasis that every $q \in R(p)$ should be executed "before" p.) For this generalization of project selection, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

Exercise 19.5. Let $G = (V, E)$ be an undirected graph with positive edge weights $w : E \to \mathbb{R}_{>0}$. We define the *weighted density* of a graph as the total edge weight divided by the number of vertices. Consider the problem of computing the weighted densest subgraph of G, assuming the edge weights are all integral and between 1 and $\text{poly}(n)$. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

Exercise 19.6. In the playoff elimination problem (section 19.1.4), we asked whether a particular team was mathematically eliminated from making the playoffs. This allows for the home team would win all of its remaining games, and other teams to lose all of their games, which may be unrealistic when there are many games left. Suppose we assumed that:

1. No team will win more than 90% of its remaining games, rounded up. (I.e., if a team has k games left, it can win at most $\lceil .9k \rceil$ of them.)
2. Every team will win at least 10% of its remaining games, rounded down. (I.e., if a team as k games left, it must win at least $\lfloor .1k \rfloor$ of them.)

Consider the problem of deciding whether the home team can make the playoffs under the additional restrictions listed above. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

Exercise 19.7. Evacuate the building! Before Purdue can begin construction on its next glorious computer science building, at least two times size of Lawson and with at most half as many restrooms, the architectural plan has to pass a series of safety regulations. One of the most important safety requirements is having an *evacuation plan* where in the event of an emergency, one can empty all the rooms to safe locations without overcrowding any of the hallways and other pathways, and

risking a stampede. Here we model the problem of computing a safe evacuation plan (or declaring that none exists) as a graph problem, as follows.

The input consists of a directed graph $G = (V, E)$, two disjoint subsets of vertices $X, Y \subseteq V$, and integer edge labels $z : E \to \mathbb{N}$. X represents the rooms that would need to be evacuated, and Y represents the safe locations that the rooms in X must be evacuated to. (One safe location can be the destination of any number of evacuation routes.) The edges represent hallways/pathways through the building, and each edge e is annotated with an integer label $z(e)$ that signifies the number of evacuation paths it can accommodate. We define an *evacuation plan* as a set of paths from rooms in X to safety locations in Y, with one path starting from each room in X. We say that an evacuation plan is *safe* if no hallway/pathway is used more than its declared limit $z(e)$.

The problem is to either compute a safe evacuation plan or declare that no safe evacuation plan exists. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

Exercise 19.8. Recall that in chess, a *rook* (♖) is a piece that can move horizontally or vertically as far as it wants. (It cannot move diagonally.) We say that a rook on a particular square is *attacking* another chess piece at another square if the rook can be moved horizontally or vertically from its square to the square of that chess piece.

Consider the following problem which we call the "n-rooks problem". At a high level, there is an $n \times n$ board where some of the squares have been marked off as "unavailable". The remaining squares are "available". The goal is to place n rooks in the available squares so that none of the rooks are attacking each other. We call this a "non-attacking placement" of the n rooks. (While rooks cannot be placed on unavailable squares, they can still move over unavailable squares to attack other pieces on the other side of unavailable squares.)

Below we draw a non-attacking placement of 4 rooks on a 4×4 chess board. Unavailable squares are marked by `X`. Rooks are marked by ♖.

X	♖		X
♖	X	X	X
	X	♖	
X		X	♖

Formally, the input consists of an $n \times n$ bit-array $A[1..n, 1..n] \in \{\texttt{true}, \texttt{false}\}^{n \times n}$, where $A[i, j] = \texttt{true}$ indicates that square (i, j) is available, and $A[i, j] = \texttt{false}$ indicates that square (i, j) is unavailable. The goal is to decide if there is a non-attacking placement of n rooks in the available squares. For this problem, either (a)

design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

Exercise 19.9. This problem models the following scenario. To prepare for the final, you and your classmate have decided to try to solve all n of the exercises in the textbook. Due to time constraints, you've decided to split up the problems. To try to make the total effort even, each of you went through and skimmed all the problems, and for each problem, estimated the number of minutes it would take to solve the problem. (Thus producing two lists of estimates, one for each person.) The goal is to assign each exercise to you or our classmate as to minimize the maximum time that either of you would spend solving their assigned problems. We call this problem the "optimum study allocation problem".

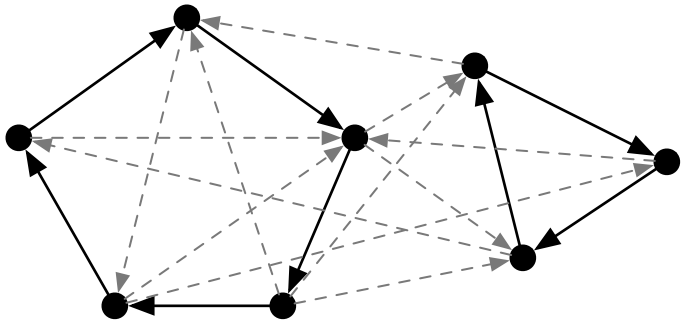
Formally, we let $\{1, \ldots, n\}$ represent the n exercises. The input consists of two lists of natural numbers $A[1..n]$ and $B[1..n]$. For $i = 1, \ldots, n$, $A[i]$ represents your estimated time for problem i and $B[i]$ represents your classmate's estimated time for problem i. The goal is to divide $\{1, \ldots, n\}$ into two sets, $S \subseteq \{1, \ldots, n\}$ and $T = \{1, \ldots, n\} \setminus S$, to minimize the maximum of

$$\sum_{i \in S} A[i] \text{ and } \sum_{j \in T} B[j].$$

Here $\sum_{i \in S} A[i]$ represents the total time you estimate to solve all the problems in S and $\sum_{j \in T} B[j]$ represents the total estimated time for your partner to solve all the problems in T.

For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

Exercise 19.10. A *cycle cover* of a given directed graph $G = (V, E)$ is a set of vertex-disjoint cycles that cover every vertex in G. You can also think of a cycle cover as a subset of edges $F \subseteq E$ such that each vertex v has exactly one edge in F entering v, and exactly one edge in F leaving v. This implies that F has exactly n edges. Below is a directed graph where the solid arcs form a cycle cover, and the dashed arcs are the remaining edges in G.



Consider the problem of deciding if a directed graph G contains a cycle cover. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.