情報数理工学実験第一／コンピュータサイエンス実験第一

J8 課題：最適化とグラフ *

Shummin NAKAYAMA

2024 年 4 月 10 日

0 事務的事項

- 教員：中山舜民 (なかやま しゅんみん)
  - 居室：西 3 号館 2 階 220 号室，E-mail：snakayama@uec.ac.jp
- アシスタント：中村 優貴（なかむら まこと）岡本研 M2
- 成績評価
  - レポートによる．
  - 1 回のレポートにより素点が付く．(注意：レポート提出をしても合格するとは限らない．)
  - レポート提出期限：2024 年 5 月 8 日 (水) 16:00（当然厳守)
  - Google classroom で課題を出題するので，そこでアップロード
- 実験は対面で西 9-135 号講義室で実施する．
  - 各自のパソコンで課題に取り組む．
  - 毎回の課題の進め方（目安）のスライドをよく読み課題に取り組むこと．
  - 実験時間中に TA と一緒に巡回しているので，質問がある場合はそこで受け付ける．

* Special thanks to Yoshio Okamoto, Masakazu Muramatsu, and Satoshi Takahashi.

1 グラフによる数理モデル化

現実世界で生起する問題の多くは離散的な視点からモデル化することにより，その本質が捉えられ，それに対する適切な解決法が導けるようになることが多い．離散モデルにおいてよく用いられる概念が「グラフ」である．グラフは対象の間の二項関係を表現する際に有用であり，関係同士が持つ関係や，関係がもたらす構造を調べる際には，グラフに関わる概念を用いると大きな助けとなる．

なお，グラフについては，1 年後学期「離散数学」 (必修) の最後の方で扱ったクラスもあるかもしれない．グラフに関する数学的側面は 3 年前学期「グラフとネットワーク」 (選択) で扱う．グラフに関するアルゴリズムは 2 年後学期「アルゴリズム論第一」 (必修) の最後の方や 3 年前学期「アルゴリズム論第二」 (選択) の前半に登場する．

それでは，例として，現実世界で起きそうな次の問題を考える．

例題 1：野球チーム構成問題

一郎，二郎，三郎，四郎，五郎，六郎，七郎，八郎，九郎の 9 人が野球チームを作ろうとしている．野球は 1 チーム 9 人で行い，9 人がそれぞれピッチャー，キャッチャー，ファースト，セカンド，サード，ショート，レフト，センター，ライトの 9 つのポジションを守る．この 9 人は次のように守ってもよいポジションに対して好みを持っている．

- 一郎はどのポジションを守ってもよいと思っている．
- 二郎はピッチャー，ファースト，セカンド，サード，ショートなら守ってもよいと思っている．
- 三郎はレフト，センター，ライトなら守ってもよいと思っている．
- 四郎はピッチャーとキャッチャーなら守ってもよいと思っている．
- 五郎はレフト，センター，ライト以外なら守ってもよいと思っている．
- 六郎はファースト，セカンド，サード，ショートなら守ってもよいと思っている．
- 七郎はピッチャー以外なら守ってもよいと思っている．
- 八郎はセカンド，サード，ショートなら守ってもよいと思っている．
- 九郎はピッチャとキャッチャー以外なら守ってもよいと思っている．

このとき，この 9 人はそれぞれが守ってもよいポジションにつくように野球チームを作ることができるだろうか？

考えるだけで頭が痛くなりそうな問題ではあるが，この問題をグラフによって解決してみたい．

グラフの数学的な定義は後に回すが，それは「頂点」と呼ばれる点と，頂点同士を結ぶ「辺」から構成される図だと思っていれば，今のところは問題がない．

この問題に対して，どのような頂点とどのような辺を使うのか考える．

まず，9 人の登場人物のそれぞれに対応する頂点を考える．そして，9 つのポジションのそれぞれに対応する頂点を考える．合わせて，18 個の頂点を考えることになる．

そして，登場人物を表す頂点とポジションを表す頂点の間に辺を結ぶのは，その人物がそのポジションを守ってもよいときで，そのときのみとする．例えば，「二郎はピッチャー，ファースト，セカンド，サード，ショートなら守ってもよいと思っている」ので，二郎に対応する頂点と，ピッチャー，ファースト，セカンド，サード，ショートに対応する 5 つの頂点がそれぞれ辺で結ばれる．その全体像を描いたのが図 1 (左) である．

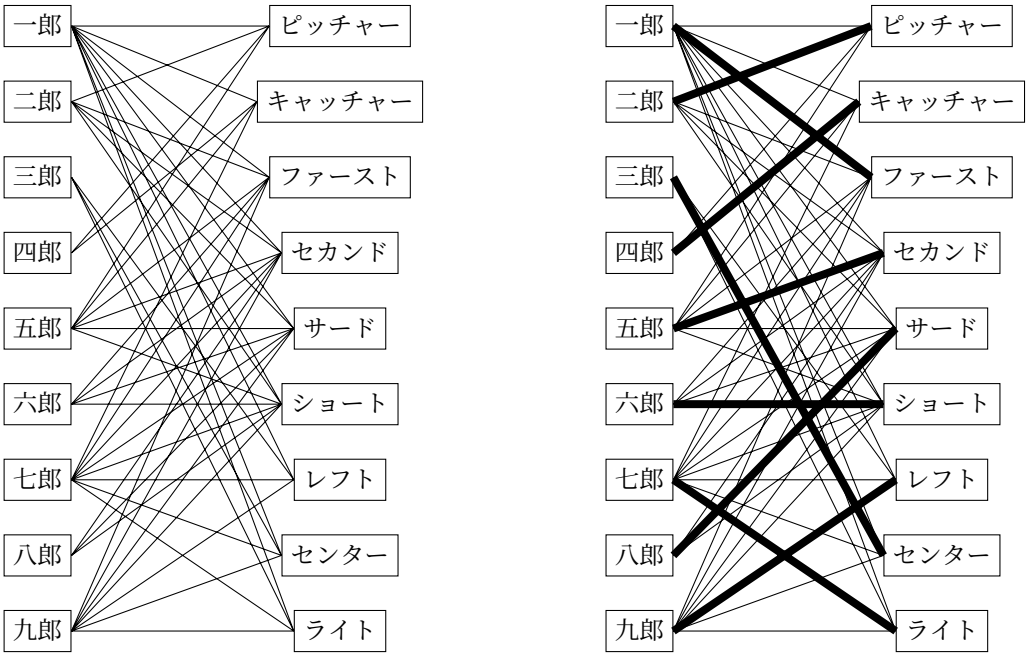


図 1　野球チーム構成問題をグラフでモデル化する.

このグラフが問題に記述されている状況を表現していることを確認してほしい.

このグラフを見て，野球チームが作れることが何に対応するのか考えてみる．登場人物はそれぞれどこかのポジションにつくが，それはこのグラフにおける辺を選ぶことに対応する．このように選ぶ辺は次の条件を満たしていないといけない.

1. 各登場人物から出ている辺の中で，選ばれる辺はちょうど 1 つである.
2. 各ポジションから出ている辺の中で，選ばれる辺はちょうど 1 つである.

別の言い方をすると，この 2 つの条件を満たすように辺を選ぶことができることと，野球チームが作れることが対応しているのである．すなわち，この 2 つの条件を満たすように辺を選ぶ，ということが考えたい問題である．実際，このグラフにおいてはそのような辺の選び方が可能であり，その 1 つが図 1 (右) に示してある．これにより，次のようにポジションを割り当てれば，野球チームが作れることが分かる.

- 一郎はファーストを守る.
- 二郎はピッチャーになる.
- 三郎はセンターを守る.
- 四郎はキャッチャーになる.
- 五郎はセカンドを守る.
- 六郎はショートを守る.
- 七郎はライトを守る.
- 八郎はサードを守る.
- 九郎はレフトを守る.

同じような雰囲気を持つ，違う問題を見てみよう．

例題 2：美容院予約問題

この美容院はオープンして 1 年足らずだが，とても評判がよい．3 カ月後の予約もほぼ埋まっている．経営者のシンディーは予約のメールをチェックするのが日課だが，いつも頭を悩ませているのは，お客さんの希望時間帯がまちまちなので，どのお客さんにいつ来てもらうのか決めることである．しかも，予約メールは多過ぎるので，すべての予約を受け付けられないかもしれないのである．ある日のメールは次のようになっていた．美容院の開店時間は朝 10:00 から夕方 17:00 までで，一人のお客さんのサービスには 1 時間かかる．(つまり，1 日にサービスできるお客さんの数は最大 7 である．) この日は 9 人のお客さんから予約メールがあり，そこには希望するサービス時間帯が書かれている．内容を要約すると次のようになる．

- アリス：13:00〜14:00，14:00〜15:00，15:00〜16:00，16:00〜17:00 のどれか．
- ベッキー：12:00〜13:00，13:00〜14:00，14:00〜15:00 のどれか．
- キャロル：10:00〜11:00，11:00〜12:00，12:00〜13:00 のどれか．
- デイジー：10:00〜11:00，12:00〜13:00，14:00〜15:00 のどれか．
- エヴァ：14:00〜15:00，15:00〜16:00 のどちらか．
- フローラ：12:00〜13:00，16:00〜17:00 のどちらか．
- グロリア：11:00〜12:00 のみ．
- ヘレン：10:00〜11:00，11:00〜12:00，16:00〜17:00 のどれか．
- アイリーン：10:00〜11:00，11:00〜12:00，12:00〜13:00 のどれか．

このとき，できるだけ多くのお客さんをサービスするためには，どのように予約を処理すればよいのだろうか？

この問題もグラフによって解決してみたい．

まず，どのような頂点を準備するか考える．野球チーム構成問題のときと同様に，9 人のお客さんのそれぞれに対応する頂点を考える．そして，お客さんをサービスできる 7 つの時間帯のそれぞれに対応する頂点を考える．合わせて，16 個の頂点を考えることになる．

そして，お客さんを表す頂点と時間帯を表す頂点の間に辺を結ぶのは，そのお客さんがその時間帯なら美容院に来られるときで，そのときのみとする．例えば，「キャロル：10:00〜11:00，11:00〜12:00，12:00〜13:00 のどれか」となっているので，キャロルに対応する頂点と 10:00〜11:00，11:00〜12:00，12:00〜13:00 に対応する 3 つの頂点がそれぞれ辺で結ばれる．その全体像を描いたのが図 2 (左) である．このグラフが問題に記述されている状況を表現していることを確認してほしい．

このグラフにおいて，図 2 (右) の太線で示したようにお客さんを各時間帯に割り当てれば，7 つの時間帯すべてでお客さんをサービスすることができる．これでシンディーの悩みも解決した．

この 2 つの例題における共通点はグラフを用いてモデル化を行っていることだけではなく，グラフにおいて発見したい部分構造が同じであるということである．それは，辺の部分集合であり，上手な割当を表現するものである．これは「マッチング」と呼ばれる概念であり，広い文脈でよく用いられる．

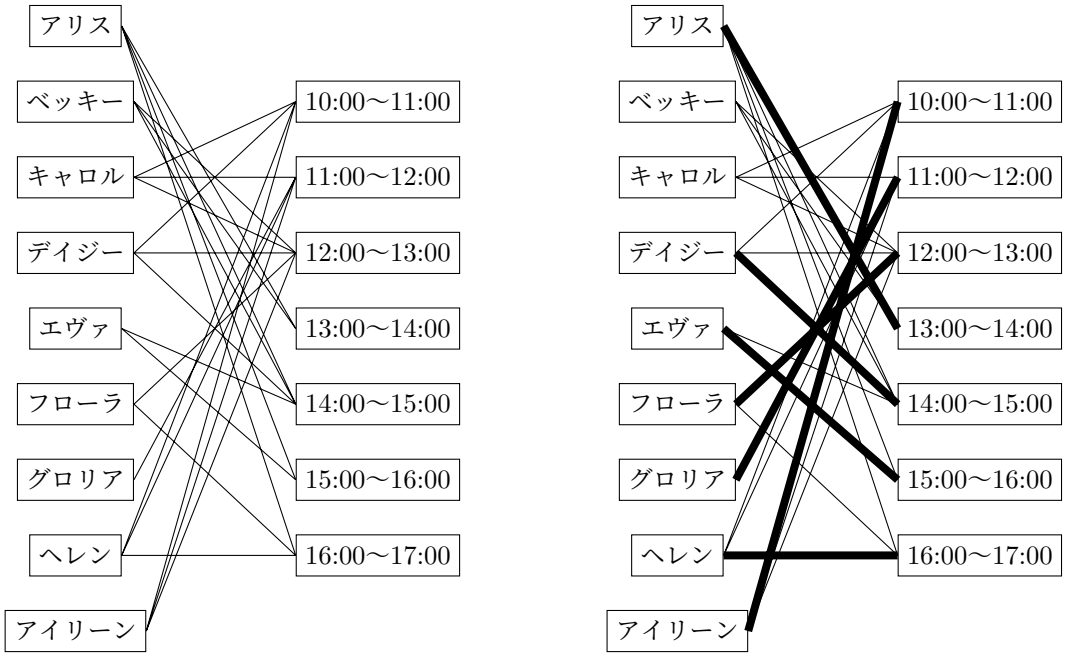


図 2　美容院予約問題をグラフでモデル化する.

1.1　問題解決の道具として数学とプログラムを利用すること

実世界における問題を数理的に解決するための道筋は通常，次のように捉えられる.

1. 実世界における問題を数理的なことばで記述する.
2. 記述された数理的対象を解析する.
3. その解析結果を実世界におけることばで解釈する.
4. これを繰り返す.

はじめのステップは「モデル化」,「定式化」と呼ばれる過程である．モデル化によって得られた数理的記述のことを「数理モデル」あるいは単に「モデル」と呼ぶ[*1].

「数理的なことば」と言っても様々な手法があり，どのような手法を用いてモデル化を行うかということは重要な選択である．また，実世界における現象を人間の観察によってモデル化するわけなので，その記述には限界がある．そのため，実世界の問題と数理モデルの間に多かれ少なかれ差異があるということを認識することが重要である.

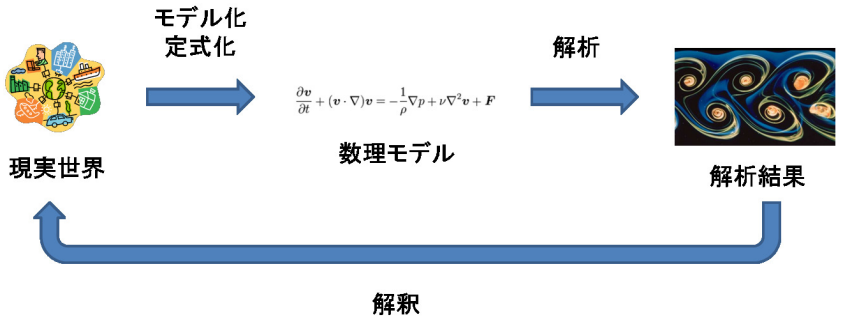
第二のステップでは数理モデルに対する解析を行うが，大きく分けて二つの手法がある．一つは伝統的な数学的な手法であり,「紙と鉛筆」によって，その数理モデルの性質を調べるというものである．もう一つは計算科学的な手法であり，すなわち数理モデルの性質を計算機によって明らかにしようとするものである．これは分野によってはシミュレーション (simulation) と呼ばれることもあるが，他の分野でシミュレーションという用語が違う意味で使われることもあるので注意が必要である．特に，実世界における現象を計算機で模倣

[*1] 数学基礎論におけるモデルとは異なる意味で用いているので注意．数学基礎論の一分野に「モデル理論」というものがあるが，それもこの実験で扱う数理モデルと直接関係を持っているわけではない.

する際にシミュレーションという語が用いられる．

第三のステップでは，今一度，解析結果を実世界の問題に照らし合わせて解釈し直す．この時点で，モデル化を行う際に見逃していた重要な側面や解析における間違い (例えば，証明の誤りやプログラムのバグ) が見つかることがある．第四のステップでは，それを考慮して，再びモデル化のステップに戻る．

次の図がこの過程のイメージである．



この流れにおいて重要な視点は，数学や計算機，プログラムを実世界における現象の理解，または，実世界における問題の解決を目的として用いるということである．これは数学そのものを探究すること，計算機そのものを探究することとは大きく違う視点であり，強調されるべきである．

2 二部グラフのマッチング

前節で登場した 2 つの例で解いている問題は「二部グラフにおける最大マッチング問題」と呼ばれるものである．ここでは，この問題を正確に述べるために，グラフやマッチングの定義を数学的に行う．

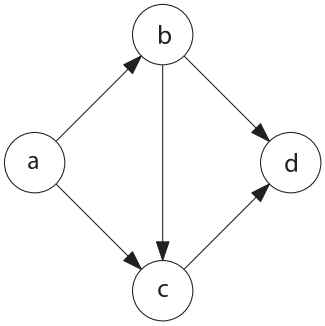
前節では，グラフのことを「頂点」と呼ばれる点と，頂点同士を結ぶ「辺」から構成される図だと思っていた．これはグラフを図形であると思うことなのだが，実をいうと，どの頂点間に辺があるのか，ということだけが重要であり，その描かれ方は本質ではない．そのため，グラフを数学的に定義する際には，どの頂点間に辺があるのか，という情報だけを取り出して，描かれ方は捨て去る．そのようにして出てくるのが，以下に挙げるグラフの定義である．

グラフ (graph) とは，ある集合 V とその集合の直積の部分集合 $E \subseteq V \times V$ の組 (V, E) として定義される．集合 V の要素をこのグラフの**頂点** (vertex) と呼び，集合 E の要素をこのグラフの**辺** (edge) と呼ぶ．

例えば，$V = \{a, b, c, d\}$，$E = \{(a, b), (a, c), (b, c), (b, d), (c, d)\}$ としたときの組 (V, E) はグラフである．

これが数学的な定義であるが，このように記述されても感覚がつかみにくいので，グラフを図示することが多い．これが先ほどの段落で捨て去った「描かれ方」になる．実際は，各頂点を点 (あるいは，大きさを持った図形) として描き，辺は対応する 2 頂点を結ぶ曲線として描く．曖昧さを排除し，読みやすく描くことは重要である．

以下の図は先ほどのグラフを描いた例である．



例えば，(a, b) という辺は a から b へ向かう矢印として表されている．今一度強調するが，この描かれたものがグラフなのではなく，グラフは集合として数学的には定義され，この描かれたものはグラフを図として表現したものである．

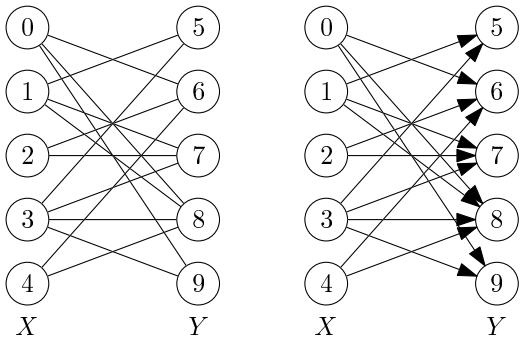
既に気付いているかもしれないが，上の図では辺が矢印として描かれているが，前節で登場したグラフの図では辺が矢印としては描かれていなかった．この実験では辺が矢印として描かれるようなグラフのみを対象とし，そのようなグラフは**有向グラフ** (directed graph) とよく呼ばれる．一方，矢印として辺を描かないグラフは**無向グラフ** (undirected graph) と呼ばれ，(u, v) という辺が存在するとき，それを (v, u) という辺と区別しない，という慣習を用いる．(これは，$V \times V$ 上のある同値関係における同値類を考えるということであるが，ここでは深入りしないことにする．)

これにより，辺は矢印として図では表現されることになる．辺 (u, v) が矢印で表現されているときのことを想像して，この辺においては u のことを始点，v のことを終点と呼ぶことが多い．

グラフ $G=(V,E)$ の頂点 $v \in V$ を考える．頂点 v の**出次数** (でじすう，out-degree) とは，G において v を始点とする辺の総数である．頂点 v の**入次数** (いりじすう，in-degree) とは，G において v を終点とする辺の総数である．

グラフ $G=(V,E)$ が**二部グラフ** (bipartite graph) であるとは，V が 2 つの集合 X,Y に分割でき，すなわち，$V=X \cup Y$ かつ $X \cap Y=\emptyset$ となる X,Y が存在して，グラフ $G=(V,E)$ のすべての辺 (u,v) が $u \in X$ かつ $v \in Y$ を満たすことである．前節で登場した，野球チーム構成問題と美容院予約問題のグラフはどちらも二部グラフであり，例えば，野球チーム構成問題においては，X が登場人物に対応する頂点全体の集合，Y がポジションに対応する頂点全体の集合である．

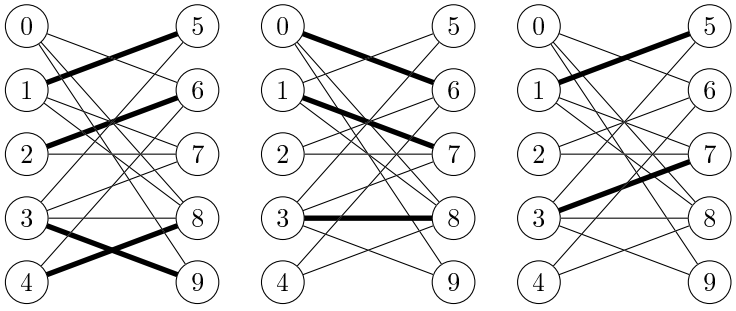
次の図のように，二部グラフではすべての辺が X の側から Y の側へ向きをつけられた矢印として描かれることに注意する．



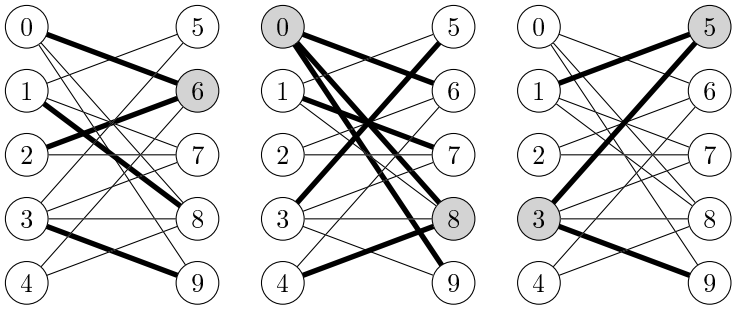
グラフ $G=(V,E)$ が二部グラフであるとき，$G=(V,E)$ と書かずに，V の分割 X,Y を明示して $G=(X,Y,E)$ と書くことがある．以後，この記法も用いていく．

二部グラフ $G=(X,Y,E)$ の**マッチング** (matching) とは，辺部分集合 $M \subseteq E$ で，任意の $u \in X$ に対して，u を始点とする M の辺が高々 1 つであり，任意の $v \in Y$ に対して，v を終点とする M の辺が高々 1 つであることである．

次の図に示された二部グラフにおいて，太線で描かれた辺の集合はどれもマッチングである．



しかし，次の図に挙げるものはマッチングではない．



それは，灰色で塗られた頂点において，それを始点 (あるいは終点) とする辺の数が 2 以上になっているからである．

前節で挙げた野球チーム構成問題と美容院予約問題で見つけようとしていたものは，二部グラフのマッチングの中で辺数が最大のものである．そのようなマッチングを最大マッチングと呼ぶ．より形式的に書くと以下のようになる．二部グラフ $G = (X, Y, E)$ のマッチング M が G の**最大マッチング** (maximum matching) であるとは，G の任意のマッチング M' に対して，M の要素数が M' の要素数以上であることである．

以上の準備に従って，目標となる，この実験で解きたい問題が以下のように記述できる．

二部グラフの最大マッチング計算問題

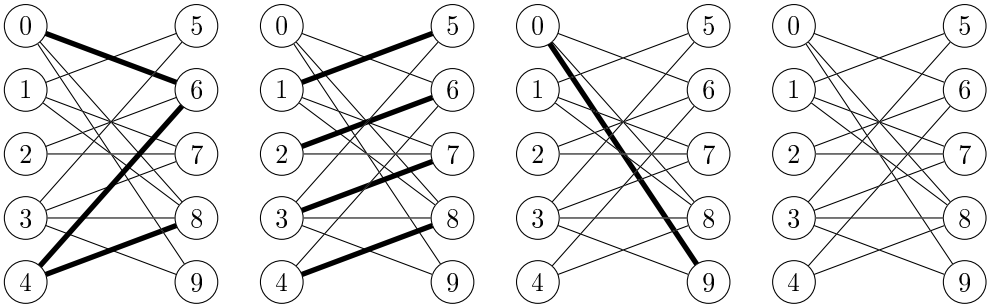
入力： 二部グラフ $G = (X, Y, E)$
出力： G の最大マッチング

では，今までの定義が理解できているか次の問題を解いて確認してもらいたい．(模範解答は存在しない．受講者間で答え合わせをして，進めてもらいたい．)

■確認問題 1 次の V を頂点集合，E を辺集合として持つ有向グラフを図示せよ．また，各頂点の出次数と入次数を答えよ．

1. $V = \{1, 2, 3, 4\}$, $E = \{(1, 3), (1, 4), (2, 3), (3, 4)\}$.
2. $V = \{0, 1, 2, 3, 4\}$, $E = \{(0, 2), (1, 2), (2, 1), (3, 4), (4, 2)\}$.

■確認問題 2 次の二部グラフにおいて，太線で示されている辺集合はマッチングであるか，そうではないか，判定せよ．



1. 2. 3. 4.

(注：問 4 では太線で示された辺が存在しない．すなわち，「太線で示されている辺集合」は空集合である．)

3 二部グラフの最大マッチングを発見するアルゴリズム

3.1 アルゴリズム設計における重要な視点と素朴な方法の問題点

二部グラフの最大マッチングを効率良く見つける方法 (アルゴリズム) を考える.
アルゴリズムを考える上で重要な点は以下の 2 つである.

アルゴリズム設計における重要項目

正当性 アルゴリズムが問題を正しく解くこと. または, 望まれる性質を必ず満たす (という保証がある) こと.

効率性 アルゴリズムの効率がよいこと. または, 効率に対する評価が (それがよい評価であれ悪い評価であれ) なされていること.

特に, 問題を正しく解くアルゴリズムを設計しようとするときには, まず効率性については考えず, 本当に問題を正しく解くアルゴリズムとして簡単に思いつくものを考えるのがよい. 特に,「力ずく」と呼ばれる手法 (brute-force,「しらみつぶし」とも呼ばれる) は様々な問題に適用可能である. 今考えている問題に対しては例えば以下のようになる.

アルゴリズム : 力ずく法

入力 : 二部グラフ $G = (X, Y, E)$
ステップ 1 : $s = 0$ // 最終的に出力される最大マッチングの要素数を初期化
ステップ 2 : for each ($M = E$ の部分集合)
　ステップ 2-1 : if (M がマッチングであり, M の要素数 $> s$) then $s = M$ の要素数
ステップ 3 : return s

このアルゴリズムはステップ 2 において E の部分集合 M をどのように見つけるかという部分にあいまいさがあるものの, それを除けば正しいアルゴリズムである. しかし, 問題点は効率の悪さである. 実際, ステップ 2 において見るべき E の部分集合 M の数は E の要素数 (グラフ G の辺数) に関して指数関数的に増加する.

では, 一方で, 効率のよさそうなアルゴリズムとして, 次のような貪欲法 (どんよくほう, greedy algorithm) を考えてみる. これは辺を任意の順番で追加していくことでマッチングを構成する方法である.

アルゴリズム : 貪欲法

入力 : 二部グラフ $G = (X, Y, E)$
ステップ 1 : $M = \emptyset$ // マッチング M の初期化
ステップ 2 : for each ($e = E$ の要素)
　ステップ 2-1 : if ($M \cup \{e\}$ がマッチングである) then $M = M \cup \{e\}$
ステップ 3 : return M の要素数

これはステップ 2 における for ループの反復回数が E の要素数しかないので, 極めて効率がよい. しかしながら, このアルゴリズムは最大マッチングを出力するとは限らない. (なぜ, 最大マッチングを出力するとは限らないのか考えてみよ.)

力ずく法は正当性を満たすが，効率性に問題があり，貪欲法は効率がよいけれども，正当性を満たさない．そのため，正当性を満たし，なおかつ，効率のよいアルゴリズムが希求される．

そのような手法としてよく用いられるものが，次に挙げる「増加道（増加路）アルゴリズム」である．

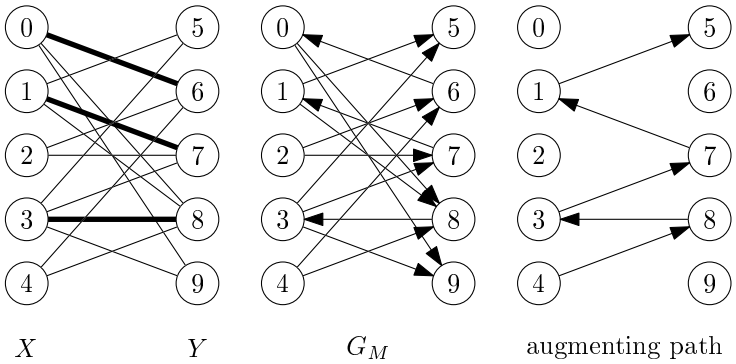
3.2 効率がよく，正しいアルゴリズム：増加道アルゴリズム

増加道アルゴリズムを説明するために，少し用語の説明が必要となる．

二部グラフ $G = (X, Y, E)$ のマッチング $M \subseteq E$ を考える．いま，二部グラフ G の辺はすべて X の頂点を始点とし，Y の頂点を終点としているが，その中で，M の要素となっている辺の向きをすべて反転する．そのようにして得られる新たな有向グラフを G_M と書くことにする．このとき，マッチング M に関する増加道 (ぞうかどう，augmenting path) とは，次を満たすグラフ G_M の頂点の列 $v_1, v_2, v_3, \ldots, v_k$ で，次を満たすもののことである．

1. $v_1, v_3, \ldots$ という奇数の添え字を持つ頂点は X の要素である．
2. $v_2, v_4, \ldots$ という偶数の添え字を持つ頂点は Y の要素である．
3. k は偶数である．
4. 任意の $i = 1, 2, \ldots, k-1$ に対して，v_i を始点とし，v_{i+1} を終点とする辺が G_M に存在する
5. グラフ G_M において頂点 v_1 の入次数は 0 である．
6. グラフ G_M において頂点 v_k の出次数は 0 である．

例えば，次の図を例とすると，左の図にある二部グラフ G において，太線で示されたマッチング M を考えると，G_M は中央の図にあるグラフとなる．



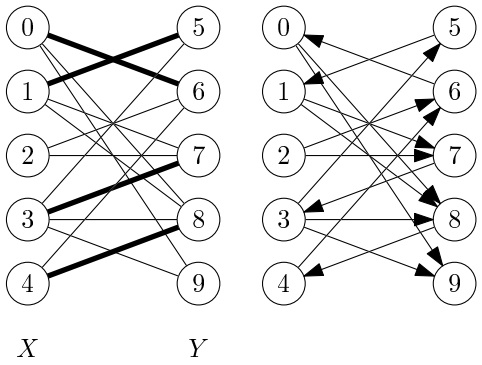
ここで，右の図に示した「$4, 8, 3, 7, 1, 5$」という頂点の列は増加道である．上の定義に沿うと，$v_1 = 4, v_2 = 8, v_3 = 3, v_4 = 7, v_5 = 1, v_6 = 5$ で $k = 6$ となっていることに注意する．では，先ほどの条件がすべて満たされていることを確認しよう．

1.「v_1, v_3, v_5 という奇数の添え字を持つ頂点は X の要素である」という条件は満たされている．実際，$v_1 = 4, v_3 = 3, v_5 = 1$ であり，どれも X の要素である．
2.「v_2, v_4, v_6 という偶数の添え字を持つ頂点は Y の要素である」という条件は満たされている．実際，$v_2 = 8, v_4 = 7, v_6 = 5$ であり，どれも Y の要素である．
3.「k は偶数である」という条件は満たされている．実際，$k = 6$ である．
4.「任意の $i = 1, 2, \ldots, k-1$ に対して，v_i を始点とし，v_{i+1} を終点とする辺が G_M に存在する」という

条件は満たされている．実際，右の図に描かれている辺はどれも中央の図に描かれている G_M の辺である．

5.「グラフ G_M において頂点 v_1 の入次数は 0 である」という条件は満たされている．実際，$v_1 = 4$ の入次数が 0 であることは中央の図を見れば分かる．
6.「グラフ G_M において頂点 v_k の出次数は 0 である」という条件は満たされている．実際，$v_6 = 5$ の出次数が 0 であることは中央の図を見れば分かる．

この例を通して，アルゴリズムの説明を行っていく．いま，G_M において，この増加道に現れる辺の向きをすべて反転させてみる (次の図の右側)．そして，Y の頂点を始点として，X の頂点を終点とする辺だけを太くしてみる (次の図の左側)．



この例ではこの太くした辺はマッチングを構成している．さらに，前の図では要素数 3 のマッチングであったのに対して，この図のマッチングの要素数は 4 になっている．つまり，「増加道に現れる辺の向きをすべて反転させる」という操作によって，要素数が 1 だけ大きなマッチングを見つけることができたのである．

実を言うと，これは常に正しく，まとめると以下のようになる．

> 任意の二部グラフ G と任意のマッチング M に対して，G_M を作成し，その中の任意の増加道を持ってくる．この増加道に現れるすべての辺の向きを反転させることで，新しいマッチングが得られ，その要素数は M の要素数よりも 1 だけ大きい．

これによって，増加道が存在するときには，そこに現れる辺の向きを反転させるという操作を繰り返すことでどんどんとマッチングの要素数を大きくすることができる．では，増加道が存在しないときには何が起きているのだろうか？ 実を言うと，そのときにはマッチングが既に最大マッチングになっている．

> 任意の二部グラフ G と任意のマッチング M に対して，G_M を作成する．そこに増加道が存在しないとき，M は G の最大マッチングである．

この事実の証明は附録 付録 A としてこの資料の最後に載せる．

この事実が正しいことを認めると，次に挙げる増加道アルゴリズムの正しさも分かる．

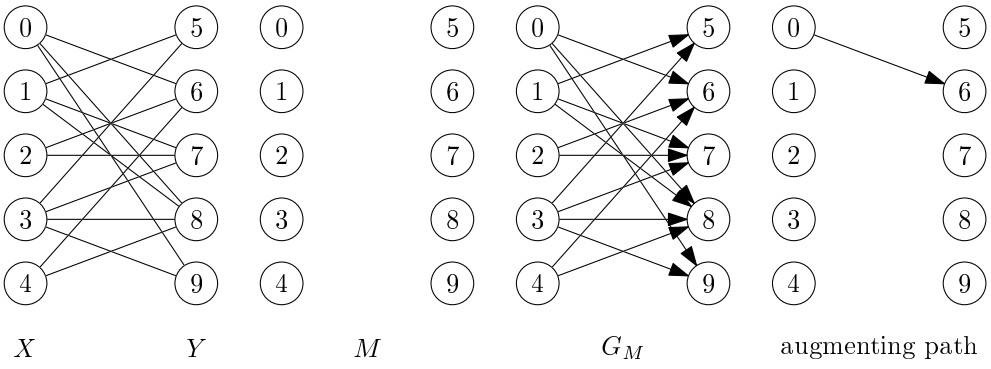
アルゴリズム：増加道アルゴリズム

入力： 二部グラフ $G = (X, Y, E)$
ステップ 1： $M = \emptyset$ // マッチング M の初期化
ステップ 2： while (G_M に増加道 P が存在)
　　ステップ 2-1： 増加道 P に現れる辺の向きをすべて反転させて G_M を作り直す
　　ステップ 2-2： Y の頂点を始点，X の頂点を終点とする辺を集めてきて，新たに集合 M を作り直す
ステップ 3： return M の要素数

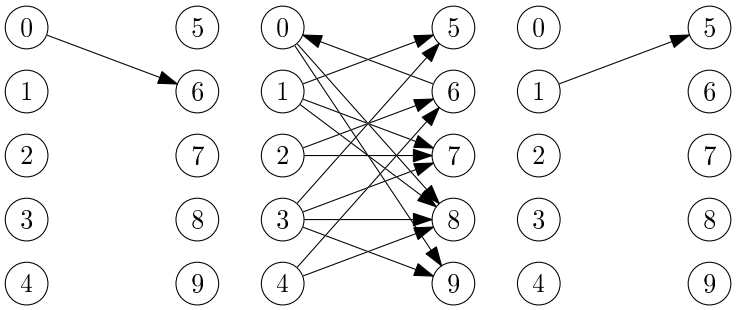
先ほどの事実を認めると，このアルゴリズムの while ループの内部が 1 回実行されると M の要素数が 1 だけ増加し，それは G_M に増加道が存在する限り続く．別の言い方をすると，増加道が存在しなくなったら while ループは終了するが，先ほどの事実から，そのときの M は最大マッチングである．したがって，while ループの反復回数は最大マッチングの要素数である．これは高々 X の要素数か Y の要素数であるので，反復回数が非常に少なく，アルゴリズムの効率も非常によいことが分かる．

アルゴリズムの動きを例によって確かめる．

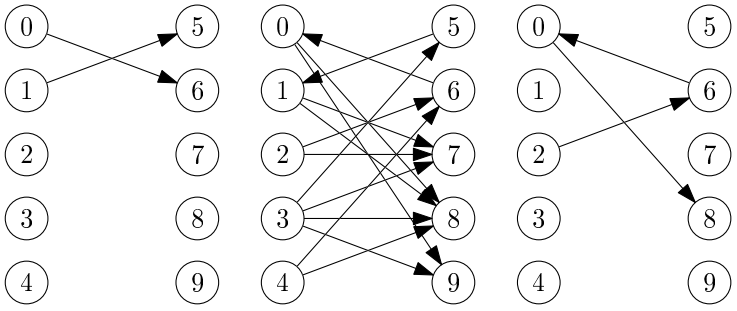
1. 下の図の最も左にあるグラフが入力として与えられたときを考える．ステップ 1 では，$M = \emptyset$ とする (左から 2 番目の図)．ステップ 2 において，G_M を作るがそれを行った結果が右から 2 番目の図である．この G_M において増加道の存在を確認する．仮に，「$0, 6$」という増加道が見つかったとする (増加道を頂点の列で表していることに注意)．これは右から 1 番目の図に示している．



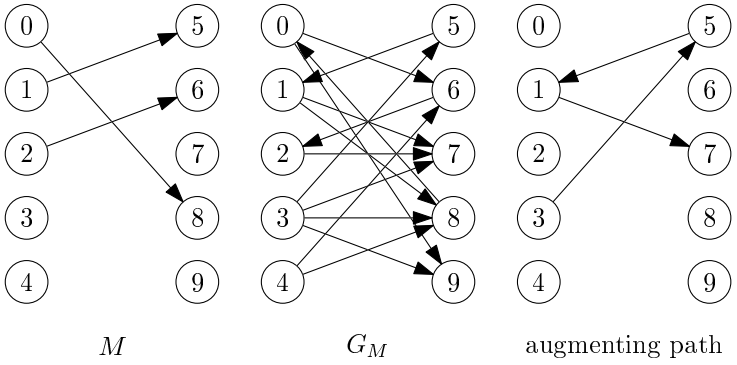
2. ステップ 2-1 で，見つかった増加道の辺を反転させて，G_M を作り直す (下の図の中央)．次にステップ 2-2 で，M も作り直す (下の図の左)．これで，ステップ 2 の冒頭に戻り，増加道の存在を確認する．仮に，「$1, 5$」という増加道が見つかったとする (下の図の右)．



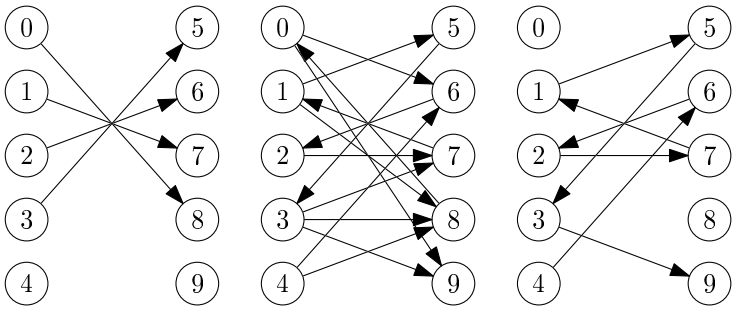
3. ステップ 2-1 で，見つかった増加道の辺を反転させて，G_M を作り直す (下の図の中央)．次にステップ 2-2 で，M も作り直す (下の図の左)．これで，ステップ 2 の冒頭に戻り，増加道の存在を確認する．仮に，「2, 6, 0, 8」という増加道が見つかったとする (下の図の右)．



4. ステップ 2-1 で，見つかった増加道の辺を反転させて，G_M を作り直す (下の図の中央)．次にステップ 2-2 で，M も作り直す (下の図の左)．いままで M の要素であった $(0, 6)$ という辺がもはや M の要素ではないことに注意する．これで，ステップ 2 の冒頭に戻り，増加道の存在を確認する．仮に，「3, 5, 1, 7」という増加道が見つかったとする (下の図の右)．

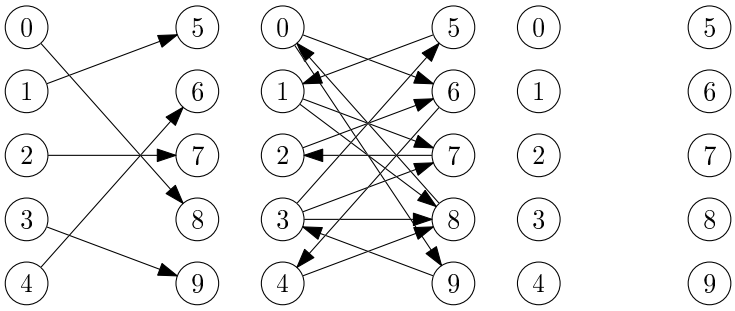


5. ステップ 2-1 で，見つかった増加道の辺を反転させて，G_M を作り直す (下の図の中央)．次にステップ 2-2 で，M も作り直す (下の図の左)．いままで M の要素であった $(1, 5)$ という辺がもはや M の要素ではないことに注意する．これで，ステップ 2 の冒頭に戻り，増加道の存在を確認する．仮に，「4, 6, 2, 7, 1, 5, 3, 9」という増加道が見つかったとする (下の図の右)．



M G_M augmenting path

6. ステップ 2-1 で，見つかった増加道の辺を反転させて，G_M を作り直す (下の図の中央)．次にステップ 2-2 で，M も作り直す (下の図の左)．先ほど M の要素でなくなった $(1,5)$ が再び M の要素になっていることに注意する．



M G_M augmenting path

これで，ステップ 2 の冒頭に戻り，増加道の存在を確認する．ここで現在の G_M には増加道が存在しないので，ステップ 2 は終了し，ステップ 3 の処理が行われる．このとき，M の要素数は 5 なので「5」が出力される．最大マッチング M 自身も発見できていることを補足する．

3.3 深さ優先探索：増加道の効率的発見

上で挙げた増加道アルゴリズムにおいて問題として残っているのは，while ループの内部を実行するかどうかを判断する「増加道の発見」をどのように行うかということである．この部分も効率良く行える必要がある．この節の残りは増加道の効率的発見法を紹介する．

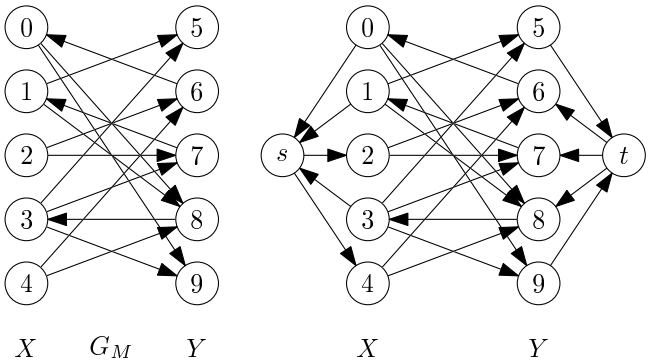
増加道は v_1 が X の頂点，v_k が Y の頂点であるような列であったことを思い出そう．増加道を見つけようとする際，その始めの頂点を X の中のどの頂点とするかという場合をいちいち考えるのが面倒であるため，それを防ぐために，新たな頂点 s を G_M に付け加えて，s と X の各点 v の間に次の規則で辺を追加する．

- G_M において v の入次数が 0 のとき，s を始点，v を終点として辺を追加する．
- G_M において v の入次数が 1 以上のとき，v を始点，s を終点として辺を追加する．

同様に，Y の方に対しても，新たな頂点 t を G_M に付け加えて，t と Y の各点 v の間に次の規則で辺を追加する．

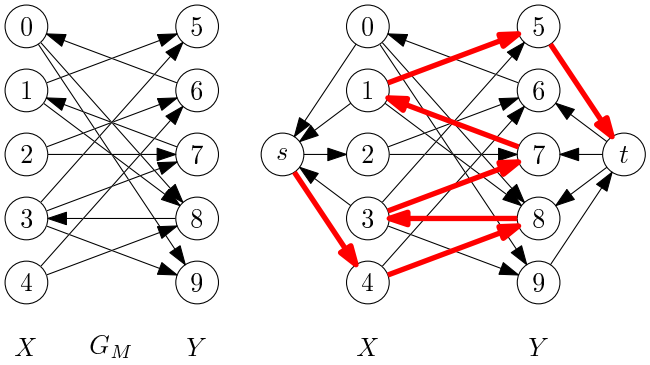
- G_M において v の出次数が 0 のとき，v を始点，t を終点として辺を追加する．
- G_M において v の出次数が 1 以上のとき，t を始点，v を終点として辺を追加する．

その結果としてできるのが下の右図にある有向グラフである．



以下の説明では，追加した頂点 s を「ソース」，頂点 t を「シンク」と呼ぶことがある．

このとき，s から t に至る有向道 (矢印の向きに沿って辺をたどることで s から t に行く道程) から M に関する増加道が得られ，その逆も成り立つことが分かる．



増加道アルゴリズムでは増加道の辺の向きを反転する操作を行うが，その代わりに，今見つけた有向道上の辺の向きをすべて反転すればよい．

一方，そのような有向道が存在しないときは，M に関する増加道がないことになる．

すなわち，増加道を見つけるために解きたい問題は次のものである．

到達可能性判定問題

入力： (二部グラフとは限らない) グラフ $G = (V, E)$ とその 2 頂点 $s, t \in V$.

出力： s から t に至る有向道があるか判定し，ある場合には，その有向道を出力する．無い場合には，「ない」ということを知らせる出力を行う．

この到達可能性判定問題を解くための標準的なアルゴリズムに深さ優先探索 (depth-first search) と呼ばれるものがある．

実を言うと，深さ優先探索においては，s から他のすべての頂点 v への到達可能性を判定する．アルゴリズムでは実際に s をスタート地点とする有向道を s の方から順に構成していく．そのための手法は再帰であり，s を始点とする辺を通り，その辺の終点 v に到達したら，v を新たな始点であると思ってアルゴリズムを再帰

的に呼び出すのである．しかし，これだけでは同じ頂点を何度も通ってしまうかもしれないので，一度訪れた頂点には印をつけて，二度と訪れることがないようにする．このとき，s を v の先行頂点 (predecessor) と呼ぶことにする．アルゴリズム実行終了時の探索結果が根つき木となることから，先行頂点を親 (parent) と呼ぶことも多い．

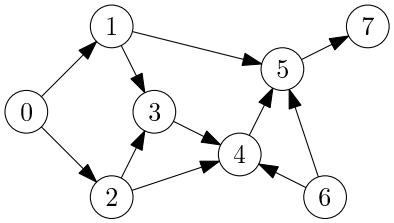
アルゴリズムの中で，各頂点に付けられる「訪問済」かどうかの印と各頂点の先行頂点は別の変数として保持する．

アルゴリズム：深さ優先探索

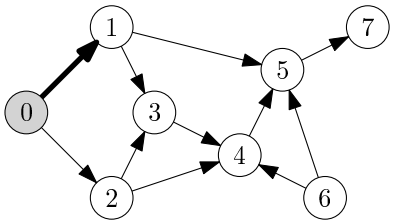
入力： (有向) グラフ $G = (V, E)$，頂点 $s \in V$
ステップ 1： s に「訪問済」という印をつける
ステップ 2： for each (v：s を始点とする辺の終点で，訪問済でないもの)
　　ステップ 2-1： v の先行頂点を s とする
　　ステップ 2-2： グラフ G，頂点 v を入力として深さ優先探索を (再帰的に) 実行
ステップ 3： return

アルゴリズムの動作例を以下に示す．入力のグラフを G，頂点を s とするときの実行を DFS(G, s) と書くことにする．

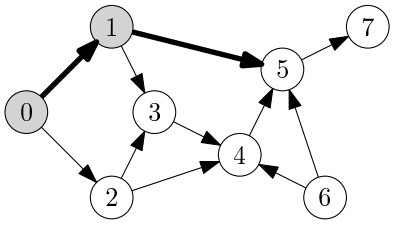
1. 次のようなグラフ G とその頂点 0 が入力として与えられたとする．



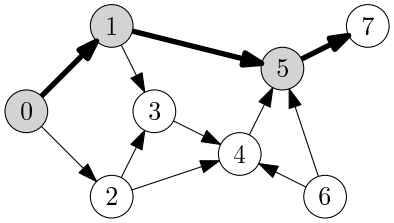
2. DFS(G, 0) の開始．ステップ 1 で 0 に訪問済という印をつける (図では灰色で示している)．ステップ 2 の条件部にある「0 を始点とする辺の終点で訪問済でないもの」を 1 つ見つける．頂点 1 が見つかったとする．ステップ 2-1 で 1 の先行頂点を 0 とする (図では太い矢印で先行頂点であるという関係を示している)．そして，ステップ 2-2 にある通り DFS(G, 1) を実行する．



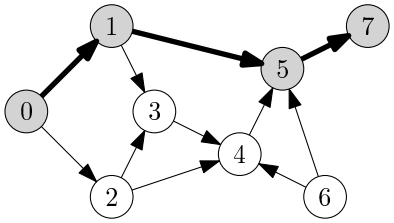
3. DFS(G, 0) の中で DFS(G, 1) の開始．ステップ 1 で 1 に訪問済という印をつける．ステップ 2 の条件部にある「1 を始点とする辺の終点で訪問済でないもの」を 1 つ見つける．頂点 5 が見つかったとする．ステップ 2-1 で 5 の先行頂点を 1 とする．そして，ステップ 2-2 にある通り DFS(G, 5) を実行する．



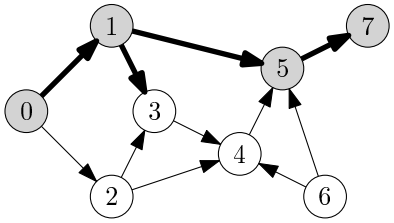
4. DFS(G, 1) の中で DFS(G, 5) の開始．ステップ 1 で 5 に訪問済という印をつける．ステップ 2 の条件部にある「5 を始点とする辺の終点で訪問済でないもの」を 1 つ見つける．頂点 7 が見つかる．ステップ 2-1 で 7 の先行頂点を 5 とする．そして，ステップ 2-2 にある通り DFS(G, 7) を実行する．



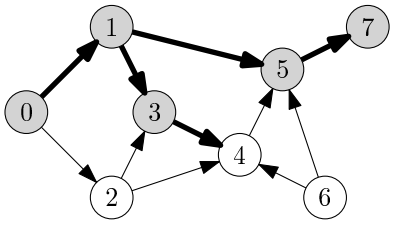
5. DFS(G, 5) の中で DFS(G, 7) の開始．ステップ 1 で 7 に訪問済という印をつける．ステップ 2 の条件部にある「7 を始点とする辺の終点で訪問済でないもの」を 1 つ見つけようとするが，それは存在しない．よって，ステップ 3 に進んで DFS(G, 7) は終了．



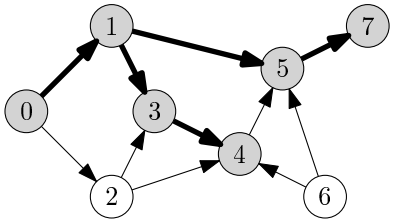
6. DFS(G, 5) のステップ 2 に戻る．ステップ 2 の条件部にある「5 を始点とする辺の終点で訪問済でないもの」を 1 つ見つけようとするが，それは存在しない．よって，ステップ 3 に進んで DFS(G, 5) は終了．
7. DFS(G, 1) のステップ 2 に戻る．ステップ 2 の条件部にある「1 を始点とする辺の終点で訪問済でないもの」を 1 つ見つける．頂点 3 が見つかる．ステップ 2-1 で 3 の先行頂点を 1 とする．そして，ステップ 2-2 にある通り DFS(G, 3) を実行する．



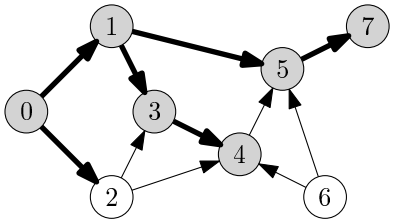
8. DFS(G, 1) の中で DFS(G, 3) の開始．ステップ 1 で 3 に訪問済という印をつける．ステップ 2 の条件部にある「3 を始点とする辺の終点で訪問済でないもの」を 1 つ見つける．頂点 4 が見つかる．ステップ 2-1 で 4 の先行頂点を 3 とする．そして，ステップ 2-2 にある通り DFS(G, 4) を実行する．



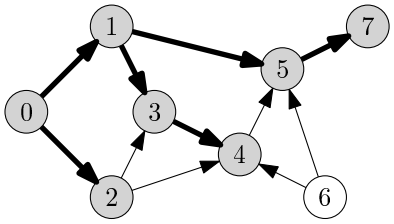
9. DFS(G, 3) の中で DFS(G, 4) の開始．ステップ 1 で 4 に訪問済という印をつける．ステップ 2 の条件部にある「4 を始点とする辺の終点で訪問済でないもの」を 1 つ見つけようとするが，それは存在しない．よって，ステップ 3 に進んで DFS(G, 4) は終了．



10. DFS(G, 3) のステップ 2 に戻る．ステップ 2 の条件部にある「3 を始点とする辺の終点で訪問済でないもの」を 1 つ見つけようとするが，それは存在しない．よって，ステップ 3 に進んで DFS(G, 3) は終了．
11. DFS(G, 1) のステップ 2 に戻る．ステップ 2 の条件部にある「1 を始点とする辺の終点で訪問済でないもの」を 1 つ見つけようとするが，それは存在しない．よって，ステップ 3 に進んで DFS(G, 1) は終了．
12. DFS(G, 0) のステップ 2 に戻る．ステップ 2 の条件部にある「0 を始点とする辺の終点で訪問済でないもの」を 1 つ見つける．頂点 2 が見つかる．ステップ 2-1 で 2 の先行頂点を 0 とする．そして，ステップ 2-2 にある通り DFS(G, 2) を実行する．



13. DFS(G, 0) の中で DFS(G, 2) の開始．ステップ 1 で 2 に訪問済という印をつける．ステップ 2 の条件部にある「2 を始点とする辺の終点で訪問済でないもの」を 1 つ見つけようとするが，それは存在しない．よって，ステップ 3 に進んで DFS(G, 2) は終了．



14. DFS(G, 0) のステップ 2 に戻る．ステップ 2 の条件部にある「0 を始点とする辺の終点で訪問済でないもの」を 1 つ見つけようとするが，それは存在しない．よって，ステップ 3 に進んで DFS(G, 0) は終了．

最終結果は上の最後の図の通り．見てわかるように，頂点 6 は頂点 0 から到達可能ではなく，アルゴリズムもそれを正しく判断している．また，頂点 0 から到達可能な頂点に対しては，その道筋 (有向道) が太線で示されている．

4 実装

この実験では以上の内容を C 言語により実装する．実装する内容は大きく分けて 3 つの部分になる．

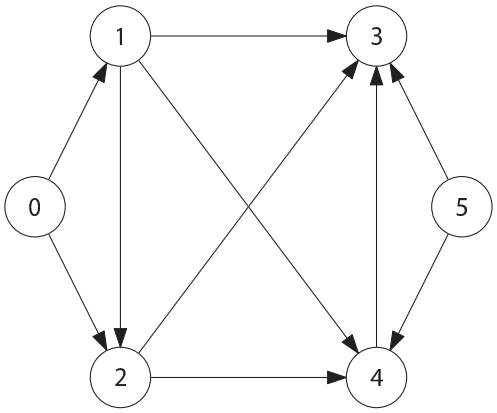
1. グラフを表現するデータ構造.
2. 深さ優先探索アルゴリズム (増加道を探索するために用いる).
3. 増加道アルゴリズム.

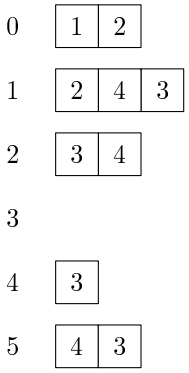
4.1 実装：グラフのデータ構造

グラフに関するアルゴリズムを実装するためには，まず，グラフをどのようにして計算機上で表現するのか定める必要がある．その方法にも様々なものがあるが，ここでは**隣接リスト** (adjacency list) と呼ばれる表現法を用いる．

グラフ $G = (V, E)$ の各頂点 $v \in V$ に対して，それを始点とする辺をリストとして保持する．辺を保持することは，その終点を保持することとして代用してもよい．リストで保持される辺はどの順序に従って並べても構わない．

例えば，次の左図にあるグラフに対して，隣接リストは右図のようになる．





頂点 3 に対するリストは空である．これは頂点 3 を始点とする辺が存在しないことに対応する．

グラフは構造体で実装する．まず，ヘッダファイル `graph_param.h` で頂点数の最大値を表す記号定数 `MAXV` と頂点の出次数の最大値を表す記号定数 `MAXDEGREE` を定義する．

ファイル：`graph_param.h`

```
#define MAXV     10000
#define MAXDEGREE  500
```

グラフ自体はヘッダファイル `graph.h` で定義する．構造体 `graph` のメンバ変数として，隣接リストそのものが `edges` に対応し，その他には各頂点の出次数を格納する `degree`，グラフの頂点数を格納する `nvertices`，グラフの辺数を格納する `nedges` を用意する．リストは配列で実装する．

ファイル：graph.h

```
typedef struct {
  int edges[MAXV][MAXDEGREE];
  int degree[MAXV];
  int nvertices;
  int nedges;
} graph;
```

グラフの頂点は 0 から nvertices - 1 の間の整数に対応するとする．

グラフのデータはファイルから読み込むこととして，その形式は以下のようになっているとする．

ファイル：kadai1/input.txt

```
8 11
0 1
0 2
1 3
1 5
2 3
2 4
3 4
4 5
5 7
6 4
6 5
```

まず，1 行目はグラフの頂点数と辺数が 1 つの空白によって仕切られて書かれている．それに続くのは，辺を表す行であり，各行が 1 つの辺に対応する．すなわち，辺数だけこれらの行が存在する．各行には 2 つの数字が 1 つの空白で仕切られて書かれている．1 つ目の数字がその辺の始点，2 つ目の数字がその辺の終点である．頂点は整数で表現され，0 から順にとばすことなく整数が使われているものとする．辺は順不同で並んでいる．

グラフに対して次のような操作を行う関数を実装する．なお，構造体に対して，それ自身のメンバ変数の値を書き換える操作を行うので，関数において，構造体はポインタで渡す．そのとき，メンバ変数は「->」によって参照することとなるので注意する．

1. グラフを初期化する関数 void initialize_graph(graph *g)．グラフ g を引数として，g の頂点数と辺数を 0 とし，各頂点の出次数を 0 と設定する．

ファイル：graph.h

```
void initialize_graph(graph *g)
{
  int i;
  g->nvertices = 0;
  g->nedges = 0;
  for(i = 0; i < MAXV; i++) {
    g->degree[i] = 0;
  }
  return;
}
```

2. グラフの頂点数を 1 つ増加させる関数 `void insert_vertex(graph *g)`．グラフ g の頂点数を 1 つ増加させる．グラフの頂点は 0 から `g->nvertices - 1` の間の整数であることに注意する．

ファイル：graph.h

```
void insert_vertex(graph *g)
{
  g->nvertices ++;
  if(g->nvertices > MAXV)
    fprintf(stderr, "Warning: insertion exceeds max number of vertices\n");
  return;
}
```

3. グラフに指定した辺が存在するか判定する関数 `int is_edge(graph *g, int x, int y)`．整数 x と y によって 2 頂点を指定し，グラフ g において，x を始点とし，y を終点とする辺が存在すれば 1，そうでなければ 0 を返す．

ファイル：graph.h

```
int is_edge(graph *g, int x, int y)
{
  int i;
  for(i = 0; i < g->degree[x]; i++) {
    if(g->edges[x][i] == y) {
      return 1;
    }
  }
  return 0;
}
```

4. グラフに指定した辺を追加する関数 `void insert_edge(graph *g, int x, int y)`．整数 x と y によって 2 頂点を指定し，グラフ g に x を始点，y を終点とする辺を追加する．既にそのような辺が存在

するときは追加しない．追加したときに，頂点 x の出次数が 1 だけ増加し，グラフ g の辺数も 1 だけ増加することに注意する．

ファイル：graph.h

```
void insert_edge(graph *g, int x, int y)
{
  if(is_edge(g, x, y)) {
    fprintf(stderr, "Warning: (%d, %d) already exists, no insertion is performed\n", x, y);
  } else {
    if(g->degree[x] >= MAXDEGREE) {
      fprintf(stderr, "Warning: insertion(%d, %d) exceeds max degree\n", x, y);
    }
    g->edges[x][g->degree[x]] = y;
    g->degree[x] ++;
    g->nedges ++;
  }
  return;
}
```

5. グラフのデータを読み込む関数 void read_graph(FILE *fp, graph *g)．ファイル fp から先に挙げた形式のグラフのデータを読み込み，その内容を g に入れる．これは insert_edge(g, x, y) を繰り返し実行することで達成される．

ファイル：graph.h

```
void read_graph(FILE *fp, graph *g)
{
  int i;
  int m;
  int x, y;

  initialize_graph(g);
  fscanf(fp, "%d %d\n", &(g->nvertices), &m);
  for(i = 0; i < m; i++) {
    fscanf(fp, "%d %d\n", &x, &y);
    insert_edge(g, x, y);
  }
  return;
}
```

6. グラフの内容を表示する関数 void print_graph(graph *g)．グラフ g の内容を標準出力に書き出す．

ファイル：graph.h

```
void print_graph(graph *g)
{
  int i, j;

  for (i = 0; i < g->nvertices; i++) {
    printf("%d: ", i);
      for (j = 0; j < g->degree[i]; j++) {
        printf(" %d", g->edges[i][j]);
      }
    printf("\n");
  }
  return;
}
```

7. グラフにおいて指定した辺を削除する関数 `int remove_edge(graph *g, int x, int y)`．整数 x と y によって 2 頂点を指定し，グラフ g に x を始点，y を終点とする辺があれば，それを削除する．そのような辺が存在しないときは，存在しないという内容のメッセージを標準エラー出力に書き出す．削除したときに，頂点 x の出次数が 1 だけ減少し，グラフ g の辺数も 1 だけ減少することに注意する．

ファイル：graph.h

```
void remove_edge(graph *g, int x, int y)
{
  /** ここを各自が実装する **/
  return;
}
```

8. グラフにおいて指定した辺の向きを逆転させる関数 `int reorient_edge(graph *g, int x, int y)`．整数 x と y によって 2 頂点を指定し，グラフ g に x を始点，y を終点とする辺があり，y を始点，x を終点とする辺がなければ，x を始点，y を終点とする辺を削除し，y を始点，x を終点とする辺を追加する．グラフ g に x を始点，y を終点とする辺が存在しないときは，存在しないという内容のメッセージを標準エラー出力に書き出す．グラフ g に y を始点，x を終点とする辺が存在するときには，存在するという内容のメッセージを標準エラー出力に書き出す．

ファイル：graph.h

```
void reorient_edge(graph *g, int x, int y)
{
  /** ここを各自が実装する **/
  return;
}
```

注意：グラフのデータ構造を作成するときには，頂点を削除する関数も用意するのが普通である．しかし，それを実装しようとすると，いろいろと難しいことがあるので，ここでは行わない．(この実験の範囲では頂点の削除を必要としない．) データ構造において，要素の追加に比べて，要素の削除は難しいことが多く，設計と実装に労力を割く．詳細は「プログラミング通論」と「アルゴリズム論第一」の内容を参照.

4.2 実装：深さ優先探索

深さ優先探索に用いる情報，つまり，各頂点に付けられる「訪問済」かどうかの印と各頂点の先行頂点を構造体 dfs_info として保持する.

ファイル：dfs.h

```
typedef struct {
  int visited[MAXV];
  int predecessor[MAXV];
} dfs_info;
```

ここで，visited[v] は v が訪問済かどうかを表し，これが 1 であるときには訪問済であること，0 であるときには訪問済ではないこととする (このような真理値をフラグとよく呼ぶ)．また，predecessor[v] は探索における v の先行頂点を表す．配列 predecessor の値はすべて -1 に初期化されることとなるため，探索終了時に visited[v]$= 0$ である頂点 v に対して，predecessor[v]$= -1$ となる．グラフの頂点は 0 から nvertices - 1 の間の整数に対応していたことに注意する.

深さ優先探索の実装のために次の関数を実装する.

1. 深さ優先探索に用いる情報を初期化する関数 void initialize_search(graph *g, dfs_info *d_i). グラフ g と深さ優先探索に用いる情報 d_i を引数として，g のすべての頂点 i (添え字として 0 から g->nvertices まで) に対する d_i->visited[i] の値を 0 に，d_i->predecessor[i] の値を -1 に設定する.

   ファイル：dfs.h

   ```
   void initialize_search(graph *g, dfs_info *d_i)
   {
     int i;
     for (i = 0; i < g->nvertices; i++) {
       d_i->visited[i]     = 0;
       d_i->predecessor[i] = -1;
     }
     return;
   }
   ```

2. 深さ優先探索を実行する関数 void dfs(graph *g, dfs_info *d_i, int start). グラフ g において頂点 start を始点とする深さ優先探索を行う．頂点 start が g の頂点であることは事前条件として仮定する．実行終了時に，頂点 i が訪問済であるとき，d_i->visited[i] の値は 1 であり，d_i->predecessor[i] の値は頂点 i の先行頂点であるようにする.

ファイル：dfs.h

```
void dfs(graph *g, dfs_info *d_i, int start)
{
  /** ここを各自が実装する **/
  return;
}
```

3. 先行頂点を表示する関数 `void print_predecessors(graph *g, dfs_info *d_i)`．深さ優先探索によって得られた先行頂点の情報を `d_i` から表示する．

ファイル：dfs.h

```
void print_predecessors(graph *g, dfs_info *d_i)
{
  int i;
  for(i = 0; i < g->nvertices; i++) {
    printf("%d: predecessor[%d] = %d\n", i, i, d_i->predecessor[i]);
  }
  return;
}
```

4. 頂点が訪問済みかどうか表示する関数 `void print_visited_vertices(graph *g, dfs_info *d_i)`．深さ優先探索によって得られた訪問済み頂点の情報を `d_i` から表示する．

ファイル：dfs.h

```
void print_visited_vertices(graph *g, dfs_info *d_i)
{
  int i;
  for(i = 0; i < g->nvertices; i++) {
    printf("%d: visited[%d] = %d\n", i, i, d_i->visited[i]);
  }
  return;
}
```

4.3 実装：増加道アルゴリズム

二部グラフの最大マッチングを発見する増加道アルゴリズムの実装において，入力される二部グラフを変数 `g` に保持し，発見されるマッチングを変数 `matching` に保持することにする．その過程で行われる深さ優先探索の情報は `d_i` に保持する．

増加道アルゴリズムの実装のために次の関数を実装する．

1. 入力グラフから有向グラフを構築する関数 `void construct_digraph_for_matching(graph *g)`．入力グラフ `g` が二部グラフであることは仮定し，出次数が 0 の頂点を Y の頂点とし，それ以外の頂点を

X の頂点とする．すなわち，グラフの辺はすべて X の頂点を始点として，Y の頂点を終点とする．このとき，この関数では新たな頂点を 2 つ追加する．それらを source，sink と呼ぶ．そして，source を始点として X の各頂点を終点とする辺を追加し，また，Y の各頂点を始点として sink を終点とする辺を追加する．ただし，どの辺の始点にも終点にもなっていない頂点は Y の頂点であると見なすことにする．実装の都合上，source の頂点番号は「g の頂点数」であるとして，sink の頂点番号は「g の頂点数 +1」であるとする．実行終了後，g の内容が構築された有向グラフとなり，それは入力時よりも多く頂点と辺を持つグラフとなっている．

ファイル：bipmatching.h

```
void construct_digraph_for_matching(graph *g)
{
  int i, source, sink;
  source = g->nvertices; insert_vertex(g);
  sink   = g->nvertices; insert_vertex(g);
  for(i = 0; i < g->nvertices -2; i++) {
    if(g->degree[i]==0) insert_edge(g, i, sink);
    else                insert_edge(g, source, i);
  }
  return;
}
```

2. 増加道に沿って辺の向きを反転させる関数 void augment(graph *g, dfs_info *d_i, graph *matching, int start, int end)．グラフ g において頂点 start から頂点 end が到達可能であるとき，start から end に至る有向道の辺の向きをすべて反転させる．増加道の探索のために深さ優先探索を用い，その情報は d_i に保持される．つまり，増加道は d_i->predecessor[] によって記述され，増加道において頂点 i の先行頂点は d_i->predecessor[i] になっている．グラフ matching は現在保持しているマッチングを表している．
   深さ優先探索によって得られた情報 d_i->predecessor[] から，頂点 start から end に至る有向道が以下のように分かる．
   (a) 説明を簡単にするために，頂点を表す変数 v と p を用意する．気持ちは，「頂点 v の先行頂点を p で表す」ということである．
   (b) まず，v = end, p = d_i->predecessor[v] とする．
   (c) v が start となるまで以下を繰り返す．
      - v = p, p = d_i->predecessor[v] とする．

   (この部分は，図を描きながら読んで，理解するように．「図を描きながら読む」というのは重要な習慣なので励行するように．)
   これによって，反復における v として，頂点 start から end に至る有向道の上の頂点が逆順に 1 つずつ得られる．
   この関数の実行終了時に，頂点 start から end に至る有向道のすべての辺が g において反転され，matching においては，その有向道上の start と end に接続しないすべての辺 (u, v) について以下が

行われる．(この部分も，図を描きながら読んで，理解するように．)

- matching に頂点 u と頂点 v を結ぶ辺が存在しない場合，つまり，(u,v) も (v,u) も matching の辺ではない場合，(u,v) が matching に挿入される．
- matching に頂点 u と頂点 v を結ぶ辺が存在する場合，u は Y の要素であり，v は X の要素であり，matching に存在するそのような辺は (v,u) である．つまり，(v,u) を matching から削除する．

以上を踏まえて関数を実装する．

ファイル：bipmatching.h

```
void augment(graph *g, dfs_info *d_i, graph *matching, int start, int end)
{
  /** ここを各自が実装する **/
  return;
}
```

3. 増加道アルゴリズムによって最大マッチングを発見する関数 int find_maximum_matching (graph *g, graph *matching)．関数実行開始時に，グラフ g は入力グラフにソースとシンクを追加したもの，グラフ matching は入力グラフと同じ頂点集合を持ち，辺を持たないグラフとする．実行終了時には，matching に発見された最大マッチングがグラフとして保持される．整数型の返り値は matching の辺の数である．

   関数の内部では，整数型の変数 size，source，sink を用い，size は返り値となる matching の辺の数を表し，source，sink はそれぞれ入力グラフに付け加えられたソースとシンクを表す．関数 void construct_digraph_for_matching(graph *g) の動きより，source は g->nvertices-2，sink は g->nvertices-1 である．アルゴリズムでは，グラフ g において source から sink への到達可能性を深さ優先探索により調べ，到達可能であったら，増加道が見つかったことになるので，その増加道に沿って関数 augment を実行する．そうでないときは，増加道が見つからなかったわけなので，最大マッチングが既に見つかっていることとなり，実行を終了する．関数の中で深さ優先探索を行うため，そのための情報を保持する d_i も用いる．頂点 sink に到達可能であるかどうかは，d_i->visited[sink] か d_i->predecessor[sink] を見れば分かる．

ファイル：bipmatching.h

```
int find_maximum_matching(graph *g, graph *matching)
{
  int size = 0; /* the size of a current matching */
  int source;
  int sink;
  dfs_info *d_i;

  source = g->nvertices -2;
  sink = g->nvertices -1;
  d_i = (dfs_info*)malloc(sizeof(dfs_info));

  /** ここを各自が実装する **/

  return size;
}
```

これらの関数を実装したとき，メイン関数を以下の通りにすれば，標準出力に発見されたマッチングとその辺数が出力される．

ファイル：kadai3main.c

```
int main(int argc, char* argv[])
{
  int size_matching; /* the size of a matching */
  FILE *fp;
  graph *g; /* a graph to analyze */
  graph *matching; /* a  matching to be found*/
  fp = fopen(argv[1], "r");
  if(fp == NULL) {
    fprintf(stderr, "File open failed\n"); fclose(fp);
    return 0;
  }
  g = (graph*)malloc(sizeof(graph));
  matching = (graph*)malloc(sizeof(graph));
  read_graph(fp, g); fclose(fp);

  initialize_graph(matching);
  matching->nvertices = g->nvertices;
  construct_digraph_for_matching(g);
  size_matching = find_maximum_matching(g, matching);

  print_graph(g); printf("\n"); print_graph(matching);
  printf("the maximum matching size = %d\n", size_matching);

  free(g); free(matching);
  return 0;
}
```

5 実験課題

5.0 実験準備

1. J8 課題に関するファイル一式が Google classroom の「授業資料」→「J8 課題プログラムファイル」にあるので，それを各自ダウンロードする．
2. 以後，このディレクトリ `mics1-j8` で作業を行うことを推奨するが，ディレクトリの名前は別のものでもよい．

5.1 課題 1 (必須課題)

■課題内容． ファイル `graph.h` における次の 2 つの関数を実装せよ．

- 関数 `void remove_edge(graph *g, int x, int y)`．
- 関数 `void reorient_edge(graph *g, int x, int y)`．

■課題詳細． 動作確認のためのプログラムが `kadai1main.c` として用意されている．コンパイルは「`gcc kadai1main.c`」で行える．

プログラム `kadai1main.c` の main 関数は，第 1 引数として，入力グラフを格納したファイル名を指定するように書かれている．例えば，入力グラフが `kadai1/input.txt` というファイルに格納されている場合は「`./a.out kadai1/input.txt`」と実行すればよい．(実行ファイルが `a.out` でない場合は，対応する実行ファイル名に `a.out` を置き換えればよい．)

動作確認のために，ディレクトリ `kadai1` にはグラフの例が 1 つ，ファイル `input.txt` として用意されている．プログラム `kadai1main.c` の main 関数の動きとファイル `input.txt` の中で記述されているグラフの内容を理解して，各自の実装が正しいかどうかを確認せよ．

■レポート記載内容． 以下の内容を順にまとめよ．

1. `void remove_edge(graph *g, int x, int y)` と `void reorient_edge(graph *g, int x, int y)` のソースコード．
2. `kadai1` にある `input.txt` を入力としたときの実行結果．
3. 実行結果の検討 (プログラムが正しく動作することをどのように確認したか？)．

5.2 課題 2 (必須課題)

■課題内容． ファイル `dfs.h` において関数 `void dfs(graph *g, dfs_info *d_i, int start)` を実装せよ．

■課題詳細． 動作確認のためのプログラムが `kadai2main.c` として用意されている．コンパイルは「`gcc kadai2main.c`」で行える．

プログラム `kadai2main.c` の main 関数は，第 1 引数として，入力グラフを格納したファイル名を指定する

ように書かれている．例えば，入力グラフが kadai2/graph0_input.txt というファイルに格納されている場合は「./a.out kadai2/graph0_input.txt」と実行すればよい．(実行ファイルが a.out でない場合は，対応する実行ファイル名に a.out を置き換えればよい．)

動作確認のために，ディレクトリ kadai2 にはグラフの例がいくつかファイルとして用意されている．入力グラフを格納するファイルには graph?_input.txt という名前がつけられ，graph?_output.txt というファイルには，そのグラフに対してプログラムを実行した後，visited[] が取る値を並べている．(この「?の部分は 0 から 9 までの数字が 1 つ入る．) プログラム kadai2main.c の main 関数の動きとそのファイルの中で記述されているグラフの内容を理解して，各自の実装が正しいかどうかを確認せよ．

■レポート記載内容． 以下の内容を順にまとめよ．

1. void dfs(graph *g, dfs_info *d_i, int start) のソースコード．
2. kadai2 にあるファイルを入力としたときの実行結果．kadai2 にあるファイルのうち 3 つを選んで実行結果を載せる．
3. 実行結果の検討 (プログラムが正しく動作することをどのように確認したか？)．

5.3 課題 3 (必須課題)

■課題内容． プログラム bipmatching.h における次の 2 つの関数を実装せよ．

- 関数 void augment(graph *g, dfs_info *d_i, graph *matching, int start, int end)．
- 関数 int find_maximum_matching(graph *g, graph *matching)．

■課題詳細． 動作確認のためのプログラムが kadai3main.c として用意されている．コンパイルは「gcc kadai3main.c」で行える．

プログラム kadai3main.c の main 関数は，第 1 引数として，入力グラフを格納したファイル名を指定するように書かれている．例えば，入力グラフが kadai3/bigraph0_input.txt というファイルに格納されている場合は「./a.out kadai3/bigraph0_input.txt」と実行すればよい．(実行ファイルが a.out でない場合は，対応する実行ファイル名に a.out を置き換えればよい．)

動作確認のために，ディレクトリ kadai3 にはグラフの例がいくつかファイルとして用意されている．ファイル名が bigraph?_input.txt となっているファイルは入力とするグラフの例であり，そのグラフの最大マッチングの要素数は bigraph?_output.txt に収められている．また，example_baseball.txt は野球チーム構成問題に対応する入力グラフ，example_coiffeur.txt は美容室予約問題に対応する入力グラフである．

■レポート記載内容． 以下の内容を順にまとめよ．

1. void augment(graph *g, dfs_info *d_i, graph *matching, int start, int end) と int find_maximum_matching(graph *g, graph *matching) のソースコード．
2. kadai3 にあるファイルを入力としたときの実行結果．kadai3 にあるファイルのうち 3 つを選んで実行結果を載せる．
3. 実行結果の検討 (プログラムが正しく動作することをどのように確認したか？)．

5.4 課題 4 (選択課題)

■課題内容. 各自が実装したプログラムの性能を測定せよ.

■課題詳細. ディレクトリ kadai4 には二部グラフをランダムに生成するプログラム randgen.c が用意してある. 使い方は以下の通りである.

- コンパイル

```
> gcc randgen.c -o randgen
```

- 実行

```
> ./randgen 20 30 0.4
```

このプログラムは引数を 3 つ取る. 1 つ目の引数は二部グラフ $G = (X, Y, E)$ における集合 X の要素数を表す. 2 つ目の引数は集合 Y の要素数を表す. 3 つ目の引数は 2 頂点間に辺が存在する確率を表す. このプログラムではこの確率に従って, X の 1 要素と Y の 1 要素間を独立に辺で結ぶ.

このプログラムが生成したグラフを入力として, 各自の作成したプログラムがどれだけの時間を計算に費やしたか測定する.

以下, 実験結果をまとめる際の注意である.

- UNIX 系 OS において, コマンドの実行時間を測定するコマンドは time である. 実行するコマンドの前に time と挿入すればよい. 例えば, 次のようなコマンドを入力して表示される結果を見てみる[*2].

```
> time ls -l
total 100
中略

0.003u 0.000s 0:00.01 0.0%      0+0k 0+0io 0pf+0w
```

最後の 1 行に時間のようなものが表示されるが, 見るべき部分は「0.003u」と書かれた項目である. そこに書かれた数字がそのコマンド自体を実行するのに消費された時間であり, この場合は 0.003 秒であったことを意味する.

- プログラムは機械的な処理を何度も反復することが得意であるのに, この課題において, 我々は同じような実行を何度も手作業で繰り返しているかもしれない. しかし, シェルスクリプトを使えば, そのような作業も自動化できる. CED の標準環境のシェルは tcsh であるので, 適当な情報源を参照しながら, 作業の効率化を行うことを薦める.
- 得られたデータを gnuplot でプロットしてみる. gnuplot の使い方については適当な情報源を参照すること. (データを可視化するためのツールとして, gnuplot 以外のものを用いてもよい.) 頂点数の

[*2] 詳しい人向け : CED の標準シェルである tcsh に入っている time が走る, ということに注意.

変化，辺数の変化，または，その他のパラメータに応じて，実行時間がどのように変化するのか，考察せよ．(「増加する」という質的な考察だけではなくて，「○○ % 増加する」のように量的な考察ができるとよい．) 測定結果の解析において，片対数プロットや両対数プロットを用いるとよいこともある．(これについては，「基礎科学実験 A」で扱っているはずである．) 適切な方法を選択すること．

■レポート記載内容． 以下の内容を順にまとめよ．

1. 行った実験内容の詳細．
2. その結果を示すプロット．
3. プロットおよびその他の解析によって得られた考察．

5.5 課題 5 (選択課題)

■課題内容． 増加道アルゴリズムにおいて増加道を探索する方法を変更し，その性能を比較せよ．

■課題詳細． 増加道の探索法としては次のものを考えることができる．

- 先に実装した探索法は深さ優先探索に基づくが，その入力として，t も指定し，t に到達できた時点で深さ優先探索を停止する．アルゴリズムは以下の通りとなる．

  アルゴリズム：終点指定型深さ優先探索

  入力： (有向) グラフ $G = (V, E)$，2 頂点 $s, t \in V$
  ステップ 1： s に「訪問済」という印をつける
  ステップ 1'： t が訪問済であれば，return
  ステップ 2： for each (v：s を始点とする辺の終点で，訪問済でないもの)
  　　ステップ 2-1： v の先行頂点を s とする
  　　ステップ 2-2： グラフ G，2 頂点 v, t を入力として深さ優先探索を (再帰的に) 実行
  ステップ 3： return

  ステップ 1' が新たに付け加えられた部分である．
- 深さ優先探索の代わりに幅優先探索 (breadth-first search) を用いる．幅優先探索は次のように記述される，キュー (queue) を用いたアルゴリズムである．キューについては「プログラミング通論」を参照．クラスによっては，「プッシュ，ポップ」の代わりに「エンキュー，デキュー」や「アペンド，ポップ」と呼んだかもしれない．

アルゴリズム：幅優先探索

入力： (有向) グラフ $G = (V, E)$，頂点 $s \in V$
ステップ 0： 空のキュー Q を用意する．
ステップ 1： s に「訪問済」という印をつけ，s を Q にプッシュする
ステップ 2： while (Q が空ではない)
　　ステップ 2-1： Q から要素を 1 つポップし，u とする
　　ステップ 2-2： for each(v：u を始点とする辺の終点で，訪問済でないもの)
　　ステップ 2-2-1： v の先行頂点を u として，v に「訪問済」という印をつける
　　ステップ 2-2-2： v を Q にプッシュする
ステップ 3： return

他にもグラフの探索法には様々な変種が存在する．調査してみよ．
　具体的には以下のように進める．

- 上に挙げた終点指定型深さ優先探索，あるいは，幅優先探索を実装せよ．(もちろん，両方実装してもよい．) それらを用いて，増加道アルゴリズムの実装を変更せよ．
- 課題 4 と同様に，実行時間を計測し，まとめよ．

■レポート記載内容． 以下の内容を順にまとめよ．

1. 行った実験内容の詳細．プログラムのソースコード．
2. その結果を示すプロット．
3. プロットおよびその他の解析によって得られた考察．

5.6 課題 6 (選択課題)

■課題内容． 各自が実装した増加道アルゴリズムと，力ずく法，貪欲法の性能を比較せよ．

■課題詳細． 具体的には以下のように進める．

- 力ずく法，あるいは，貪欲法を実装せよ．(もちろん，両方実装してもよい．)
- それぞれの実行時間，出力されるマッチングの要素数を計測せよ．
- プロットなどを通して，アルゴリズムの特色，グラフに関するパラメータに応じた依存性を比較せよ．

■レポート記載内容． 以下の内容を順にまとめよ．

1. 行った実験内容の詳細．プログラムのソースコード．
2. その結果を示すプロット．
3. プロットおよびその他の解析によって得られた考察．

6 レポート作成

6.1 レポート作成課題

- 「必須課題」はすべて行うこと．
- 「選択課題」は行っても行わなくてもよいが，行った分だけ加点する．

6.2 レポート提出形式

- PDF で提出する．
- PDF のファイル名は，「学籍番号.pdf」とすること．
- 表紙には学籍番号と氏名を明記する．
- レポート提出期限：2024 年 5 月 8 日 (水) 16:00(当然厳守)
- レポート提出場所：Google classroom

6.3 レポート提出における補足

- レポートの作成は，LaTeX を用いて行うことを推奨する．使い方については適当な情報を参照のこと．
- **「感想」はレポートに含めるべき内容ではない．**例えば，政府機関の発行するレポート (報告書，白書) に「感想」は書かれない．当然のことである．もし，「感想」を記載したい場合は，それをレポートの一部とはせず，別に提出しなければならない．レポート本体と感想を一緒にステープラでとじるのは構わないが，感想がレポートの一部ではないことに留意すること．(それは，感想の内容が採点されない，ということも意味している．)

7 最適化に関する情報源

最適化に関する和書は豊富である．例えば，以下の書籍には標準的な内容が数理，アルゴリズム，モデル化を含めて書かれている．

- 梅谷 俊治,『しっかり学ぶ数理最適化 モデルからアルゴリズムまで』，講談社，2020.
- 久野誉人，繁野麻衣子，後藤順哉,『IT Text 数理最適化』，オーム社，2012.
- 福島雅夫,『新版 数理計画入門』，朝倉書店，2011.
- 茨木俊秀,『最適化の数学』，共立出版，2011.
- 山下信雄，福島雅夫,『数理計画法』，コロナ社，2008.
- 田村明久，村松正和,『最適化法』，共立出版，2002.

本実験で扱ったネットワークや組合せ最適化に特化したものとして以下のものがある．どれもなかなか手ごたえのある本である．

- B. コルテ，J. フィーゲン (著)，浅野孝夫，平田富夫，小野孝男，浅野泰仁 (訳),『組合せ最適化 第 2

版』，丸善出版，2012.
- 繁野麻衣子，『ネットワーク最適化とアルゴリズム』，朝倉書店，2010.
- 茨木俊秀，永持仁，石井利昌，『グラフ理論―連結構造とその応用』，朝倉書店，2010.
- 室田一雄，塩浦昭義，『離散凸解析と最適化アルゴリズム』，朝倉書店，2013.

なお，本実験のプログラムは以下の書籍を参考にして作成した．

- Steven S. Skiena and Miguel A. Revilla, *Programming Challenges: The Programming Contest Training Manual*, Springer-Verlag, New York, 2003.

付録 A　増加道アルゴリズムの正当性：証明

増加道アルゴリズムが正しく最大マッチングを発見することを証明する．設定は以下の通りであった．二部グラフ G とそのマッチング M があるとき，そこから G_M という有向グラフを構成した．この G_M において M に関する増加道が存在するかどうかによって，M が最大マッチングであるかどうかが分かる，という話であった．

以前の議論から，

$$G_M \text{ において } M \text{ に関する増加道が存在するならば，} M \text{ は } G \text{ の最大マッチングではない}$$

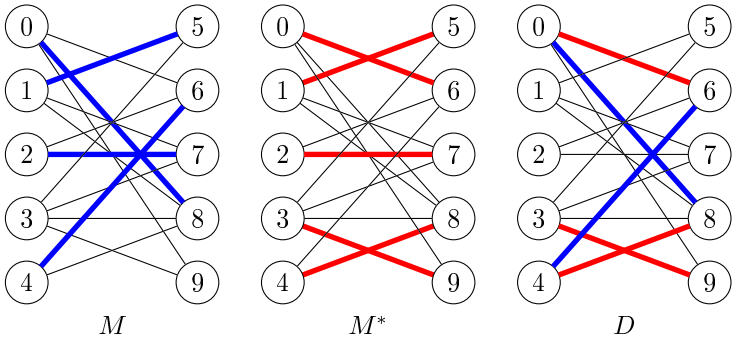
ということは分かっている．すなわち，証明したいことはこの逆であり，

$$M \text{ が } G \text{ の最大マッチングではないならば，} G_M \text{ において } M \text{ に関する増加道が存在する}$$

ということである．

この証明では，二部グラフを無向グラフとして見ていき，辺 (u, v) において，u, v はその端点と呼ばれる．頂点 v を端点とする辺の数を v の次数と呼ぶ．

では，証明のために，G の任意の最大マッチング M^* を考える．マッチング M は最大マッチングではないので，$M^* \neq M$ である．ここで考えるのは，M と M^* の中の片方にしか含まれていない辺をすべて集めてきた状況である．そのような辺全部を集めて来た集合を D とする．下の図はその例を示している．

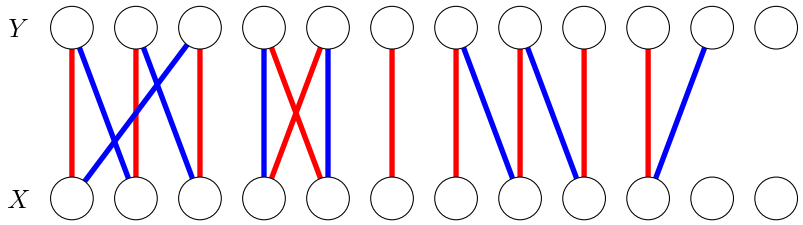


ここで D に着目すると，次の性質が分かる．

1. 集合 D というのは $(M \cup M^*) - (M \cap M^*)$ である．これは D の作り方から分かる．
2. 集合 D には M の要素と M^* の要素が混在するが，D に含まれる M の要素数は D に含まれる M^* の要素数よりも小さい．これは，$D = (M \cup M^*) - (M \cap M^*)$ であり，M の要素数が M^* の要素数よりも小さいこと (なぜなら，M^* は最大マッチングであり，M は最大マッチングではないから) ということから分かる．
3. G のどの頂点 v に対しても，それを端点とする D の要素は多くても 2 つである．これは，M がマッチングであることから，v を端点とする M の要素が多くても 1 つであり，同様に，M^* がマッチングであることから，v を端点とする M^* の要素が多くても 1 つであるので，合計して多くても 2 つということになるのである．

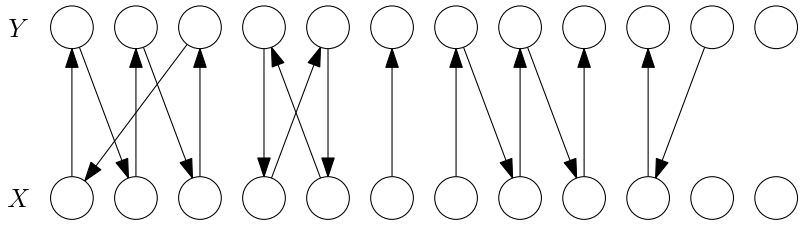
3 番目の性質から，D の要素をグラフの中に描きこんでみると，それが「サイクル」と「パス」と「孤立点」

の集まりになることが分かる．次の図では，二部グラフを 90 度回転させて描いているが，本質は今までの描き方と変わらない．



ここに現れるサイクルとパスには D の要素が含まれるが，サイクルには M の要素と M^* の要素が交互に現れるため，それらの数は同じである．しかし，上に挙げた 2 番目の性質から，D においては，M の要素数が M^* の要素数よりも小さくなくてはならない．したがって，この中のパスの (少なくとも) どれか 1 つにおいて，M の要素数が M^* の要素数よりも小さくなっている．そのようなパスにおいては M の要素と M^* の要素が交互に現れるので，M の要素数は M^* の要素数 -1 になっている．

すなわち，このパスには奇数個の辺が含まれていて，始めの辺と最後の辺はともに M^* の要素となっている．このことから，G_M においてこのパスを見てみると，M に関する増加道になっていることが分かる．



これで，「M が G の最大マッチングではないならば，G_M において M に関する増加道が存在する」ことが証明できた．