Algorithms and Data Structure
Actor Actress Matching Project

Tom Aarsen - s1027401

January 7, 2020

1 Introduction

1.1 Problem

We were tasked to find out which player has the winning strategy in the following game:

> *Veronique and Mark play the following game: Veronique starts by naming an actress x_1, then Mark responds with an actor y_1 which has co-starred with x_1, then Veronique names an actress x_2 which co-stars with y_1, and so on. Of course they are not allowed to name the same actress or actor twice. The first player who gets stuck (i.e. is not able to respond any more) loses.*
> *Given a fixed set of actresses X, a fixed set of actors Y and a fixed list of movies with casts, this is a finite game where both players have all information. So either Veronique or Mark should have a winning strategy (i.e. able to always win, no matter what moves the other player makes).*
> *Veronique always starts.*

From now on, X and Y will be used to refer to the fixed sets of actresses and actors respectively.

1.2 Input

Input to the program is given via **stdin**, in the following format:

```
2 2
DianaKruger
MelanieLaurent
BradPitt
NormanReedus
IngloriousBasterds
3
DianaKruger
MelanieLaurent
BradPitt
Sky
2
DianaKruger
NormanReedus
```

The first line contains integers n with $1 \leq n$ and m with $0 \leq m$, which represent the number of actors/actresses and the amount of movies respectively. Note that $|X| = |Y| = n$. Following this line is n lines of names of actresses, and then n lines of names of actors. Then m times the following:

- The name of the movie.
- The cast size s with $1 \leq s \leq 1000$ of the movie, as an integer.
- s lines of cast members.

All names are unique.

1.3 Output

Output should be given via **stdout**. The program should either output "Mark" or "Veronique", depending on who has the winning strategy.

2 Problem Transformation

Rather than tacking this problem as a partisan game, where we consider the options of each player, we can convert this problem as a whole to another one. Specifically, there is a mathematical relation between this problem and a Perfect Bipartite Matching (PBM). Because of this, we can attempt to find such a PBM, and use it to find out which player has the winning strategy. In this chapter I will go over the exact relation, and give a proof for this relation.

2.1 Relation

The relation between this problem and a PBM is described as follows:

$$\begin{aligned} PBM(k) &\leftrightarrow M \\ \neg PBM(k) &\leftrightarrow V \end{aligned}$$

where $PBM(k)$ is defined as *there exists a perfect matching in an undirected bipartite graph G = (X, Y, E) with X as a set of k actresses, Y as a set of k actors and E as a set of edges between them*, and M and V are predicates stating that Mark and Veronique win, respectively.
Note that $M = \neg V$, as only one of the two can win. Because of this relation between M and V, the two rules are equivalent. They are both mentioned for clarity.

2.2 Proof

2.2.1 Property 1

We start by proving the following property using induction:

$$P_1(k) \quad = \quad PBM(k) \quad \rightarrow \quad M$$

We wish to prove that this property P_1 holds for all $k \geq 1$.
To do so, we introduce a helper function $match(v)$ which returns the actor or actress with who v is matched, as long as such a match exists.

Base Case
To prove the base case, we need to prove $P_1(1)$.

1. Assume $k = 1$
2. Assume $PBM(k)$ holds.
3. (2) implies there exists an undirected bipartite graph $G = (X, Y, E)$ with $|X| = |Y| = k = 1$, which contains a perfect matching.
4. (3) and (2) imply that E contains the only possible edge between the node in X and the node in Y.
5. Veronique must pick her only actress v.
6. (4) shows that Mark can always pick his only actor u as a response to (5).
7. (5) and (1) imply that there are no unpicked actresses left.
8. (7) shows that Veronique is always unable to pick an actress in response to (6).
9. From (8) it follows that Mark wins: M holds.
10. From (1), (2) and (9) we can conclude that $P_1(1)$ holds.

This concludes the proof of the base case for proving $P_1(k)$.

Inductive Case
To prove the inductive case, we need to prove $P_1(k)$. Our inductive hypothesis (IH) for this inductive case will be that $P_1(k-1)$ holds.

1. Assume $PBM(k)$ holds.
2. (1) implies there exists an undirected bipartite graph $G = (X, Y, E)$ with $|X| = |Y| = k$, which contains a perfect matching.
3. Veronique picks some actress v.
4. From (1) it follows that a pickable actor $u = match(v)$ exists.
5. Mark picks u.
6. **Case 1:** $\neg\exists w : X[\{w, u\} \in E \wedge w \neq v]$
   (a) The case distinction condition implies that there exists **no** unpicked actress w with who Veronique can respond.
   (b) (a) implies that Mark wins: M holds.

   Case 2: $\exists w : X[\{w, u\} \in E \wedge w \neq v]$
   (a) The case distinction condition implies that there exists an unpicked actress w with who Veronique can respond.
   (b) We can construct an undirected bipartite graph representing the unpicked actors and actresses: $G' = (X - \{v\}, Y - \{u\}, \{e \mid e \in E \wedge u \notin e \wedge v \notin e\})$.
   (c) (b) implies that G' is G but with all references to u or v removed.
   (d) (b) gives that the cardinality of the sets of nodes in G' is $k - 1$.
   (e) From (c) it follows that G' contains a perfect matching similar to the one on G, but with the match between u and v removed.
   (f) From (e) and (d) it follows that $PBM(k-1)$ holds.
   (g) From (f) and the IH it follows that Mark wins: M holds.

   In both cases, Mark is guaranteed to win: M holds.
7. From (1) and (6) we can conclude that $P_1(k)$ holds.

This concludes the proof of the inductive case for proving $P_1(k)$.

This concludes the proof for $P_1(k)$ using induction. This property must hold for all $k \geq 1$.

2.2.2 Property 2

We wish to prove the following property using induction:

$$P_2(k) \quad = \quad \neg PBM(k) \quad \rightarrow \quad V$$

We wish to prove that this property P_1 holds for all $k \geq 1$.
Again, we use the helper function $match(v)$ which returns the actor or actress with who v is matched, as long as such a match exists.

Base Case
To prove the base case, we need to prove $P_2(1)$.

1. Assume $k = 1$
2. Assume $\neg PBM(k)$ holds.
3. (2) implies there does **not** exist an undirected bipartite graph $G = (X, Y, E)$ with $|X| = |Y| = k$ containing a perfect matching.
4. (3) and (2) imply that there does not exist an edge between the only node in X and the only node in Y. Hence, E is the empty set.
5. Veronique must pick her only actress v.
6. (4) shows that Mark can not pick an actor as a response to (5).
7. From (6) it follows that Veronique wins: V holds.
8. From (1), (2) and (7) we can conclude that $P_2(1)$ holds.

This concludes the proof of the base case for proving $P_2(k)$.

Inductive Case
To prove the inductive case, we need to prove $P_2(k)$. Our inductive hypothesis (IH) for this inductive case will be that $P_2(k-1)$ holds.

1. Assume $\neg PBM(k)$ holds.
2. (1) implies there does **not** exist a perfect matching on an undirected bipartite graph $G = (X, Y, E)$ with $|X| = |Y| = k$.
3. A maximum matching $MM \subseteq E$ is introduced as a set of edges each representing a matching between an actor and actress. This matching is maximum.
4. From (1) it follows that there is at least one actress who could not be matched in MM.
5. (1) shows that there is at least one actor who could not be matched in MM.
6. Veronique picks an actress v who could not be matched in MM.
7. **Case 1:** $\neg \exists u : Y[\{v, u\} \in E]$

   (a) The case distinction condition implies that there exists **no** actor u with who Mark can respond.

   (b) (a) implies that Veronique wins: V holds.

   Case 2: $\exists u : Y[\{v, u\} \in E]$

   (a) The case distinction condition implies that there exists an actor u with who Mark can respond.

   (b) u must be matched with some actress in MM, as shown by this proof by contradiction:

      i. Assume u could not be matched in MM.
      ii. From (a) it follows that $\{u, v\} \in E$.
      iii. From (6) it follows that v is unmatched.
      iv. (3) states that MM is a maximum matching. In such a matching two unmatched nodes may never have an edge between them. The edge would simply become a match.
      v. (iv) represents a contradiction. Hence, the assumption in (i) must be false. u must be matched in MM.

   (c) (b) shows that some actress $w = match(u)$ must exist and be pickable for Veronique.

   (d) We can construct an undirected bipartite graph representing the unpicked actors and actresses: $G' = (X - \{v\}, Y - \{u\}, \{e \mid e \in E \wedge u \notin e \wedge v \notin e\})$.

   (e) A maximum matching $MM' \subseteq \{e \mid e \in E \wedge u \notin e \wedge v \notin e\}$ is introduced as a set of edges each representing a matching between an actor and actress, but now for the new G'. This matching is maximum.

   (f) (d) implies that G' is G but with all references to u or v removed.

   (g) (d) gives that the cardinality of the sets of nodes in G' is $k - 1$.

   (h) There exists an actor who could not be matched in MM, who can also not be matched in MM', as shown by this contradiction:

      i. Find an actor a exists who is not matched in MM, as is shown to exist by (5).

ii. Assume this actor a can be matched in MM'.

iii. From (ii) it follows that an actress must have become available in the transition from G to G'. This actress must be w, as her match with u was dissolved when u was removed in G'.

iv. From (iii) it follows that there must be an edge $\{w, a\} \in E$. From (i) we know this $\{w, a\} \notin MM$.

v. From (a) it follows that there must be an edge $\{v, u\} \in E$. From (6) we know this $\{v, u\} \notin MM$.

vi. From (c) we know there must be an edge $\{u, w\} \in E$, and that $\{u, w\} \in MM$.

vii. (3) states that MM is a maximum matching. From (c) we know that $\{u, w\} \in MM$, but given the edges from (iv), (v) and (vi), we see that $\{v, u\} \in M$, $\{u, w\} \notin MM$ and $\{w, a\} \in MM$ would give a larger matching.

viii. (vii) contains a contradiction. Hence, the assumption in (ii) must be false. a cannot be matched in MM'.

(i) (h) shows that no perfect matching exists on G', as there is an actor who cannot be matched in G'.

(j) From (i), (f) and (g) it follows that $\neg PBM(k-1)$ holds.

(k) From (j) and the IH it follows that Veronique wins: V holds.

In both cases, Veronique is guaranteed to win: V holds.

8. From (1) and (7) we can conclude that $P_2(k)$ holds.

This concludes the proof of the inductive case for proving $P_2(k)$.

This concludes the proof for $P_2(k)$ using induction. This property must hold for all $k \geq 1$.

2.2.3 Extension

We have proven that the following properties hold:

$$\begin{array}{lcrcl} P_1(k) & = & PBM(k) & \rightarrow & M \\ P_2(k) & = & \neg PBM(k) & \rightarrow & V \end{array}$$

Now we need to extend these to show that the following holds:

$$\begin{array}{rcl} PBM(k) & \leftrightarrow & M \\ \neg PBM(k) & \leftrightarrow & V \end{array}$$

We know that $M = \neg V$. So,

$$\begin{array}{lll} & (P_1(k) \wedge P_2(k)) & = \text{ (Rewrite properties to their definitions)} \\ & (PBM(k) \rightarrow M) \wedge (\neg PBM(k) \rightarrow V) & = \text{ (Rewrite } V \text{ to } \neg M) \\ & (PBM(k) \rightarrow M) \wedge (\neg PBM(k) \rightarrow \neg M) & = \text{ } (\neg p \rightarrow \neg q \Rightarrow q \rightarrow p) \\ & (PBM(k) \rightarrow M) \wedge (M \rightarrow PBM(k)) & = \text{ (Construct equivalence using implications)} \\ \textbf{Rule 1} & PBM(k) \leftrightarrow M & = \text{ } (p \leftrightarrow q \Rightarrow \neg p \leftrightarrow \neg q) \\ & \neg PBM(k) \leftrightarrow \neg M & = \text{ (Rewrite } \neg M \text{ to } V) \\ \textbf{Rule 2} & \neg PBM(k) \leftrightarrow V & \end{array}$$

This concludes the proof. We now know we can use the existence of a perfect bipartite matching to find out whether Mark or Veronique has the winning strategy.

3 The PBM Algorithm

The program consists of several phases. The most notable two are the input parsing phase, and the matching phase. In this section I'll discuss them, as well as the data structures used.

3.1 Data classes

The following classes are created to store the parsed information. This section exists as a reference for the following sections, and differs slightly from the actual implementation purely for performance reasons.

```
class Gender:
    Male = 0
    Female = 1

class Movie(object):
    """
    Movie stores a male and female cast,
    as well as a name for this movie.
    """
    def __init__(self, name: str) -> None:
        super().__init__()
        self.name        = name
        self.male_cast   = set()
        self.female_cast = set()

```

```
    def add_cast(self, person: "Person") -> None:
        """
        Add this person either to the male or
        female cast.
        """
        if person.gender == Gender.Male:
            self.male_cast.add(person)
        else:
            self.female_cast.add(person)

class Person(object):
    """
    Person stores a name, id and gender for a person.
    It also stores a costarred set, which is a set of
    Persons of the opposite gender that played in movies
    with this person.
    There is also a match variable which stores to which
    person this person is matched in a bipartite matching.
    """
    def __init__(self, name: str, _id: int, gender: "Gender"):
        super().__init__()
        self.name      = name
        self.id        = _id
        self.gender    = gender
        self.costarred = set()
        self.match     = None

class Actress(Person):
    """
    Subclasses Person with default gender Female
    """
    def __init__(self, name: str, _id: int) -> None:
        super().__init__(name, _id, Gender.Female)

class Actor(Person):
    """
    Subclasses Person with default gender Male
    Also has explored variable used by Graph
    """
    def __init__(self, name: str, _id: int) -> None:
        super().__init__(name, _id, Gender.Male)
        self.explored = -1
```

3.2 Input Parsing

Parsing the input consists of reading all input, then fetching all actresses, followed by all actors. Afterward, for each movie, two sets (one for actors, one for actresses) partitioning the total cast of that movie are constructed. For each actress, the set of actors playing in that movie is unioned to a set of ‘costarred‘ actors, and vice versa. Lastly, the dicts of actresses and actors are converted to sets, as we no longer need the functionality from the dictionary.

```
    def parse_input(self) -> None:
        """
        Parsing input data and storing the values in the Actress
        and Actor sets.
        """
        # Read all data, and split by lines
        data = sys.stdin.read().splitlines()[::-1]
        # Get the n and m from the data
        n, m = data.pop().split()
        self.n = int(n)
        self.m = int(m)

        # Construct a dict of Actresses
        actress_dict = {}
        for i in range(self.n):
            name = data.pop()
            actress_dict[name] = Actress(name, i)

        # Construct a dict of Actors
        actor_dict = {}
        for i in range(self.n):
            name = data.pop()
            actor_dict[name] = Actor(name, i)

>       # For convenience, get a dict consisting of all people
>       people_dict = {**actress_dict, **actor_dict}
>
>       # Get all movies. For each person starring in the movie,
>       # update the person's costarred set with members they starred
>       # with in this movie, of the opposite gender.
>       for _ in range(self.m):
>           movie = Movie(data.pop())
>           for _ in range(int(data.pop())):
>               movie.add_cast(people_dict[data.pop()])
>           for person in movie.female_cast:
>               person.costarred |= movie.male_cast
>           for person in movie.male_cast:
```

```
38 >                person.costarred |= movie.female_cast
39 >
40 >        # Convert the dict to a set, leaving only the Actress
41 >        # and Actor class instances
42 >        self.actresses = {*actress_dict.values()}
43 >        self.actors    = {*actor_dict.values()}
```

3.2.1 Input Parsing Optimization

During testing, it was discovered that this parsing phase took between 3 and 1200 times as long as the matching phase. Lines 31 to 38 were discovered to be the main cause for this. To combat this, the lines prefixed with `>` were changed such that only the actresses have a set of actors they played in movies with. Unfortunately, this change prevented some possible optimization steps in the algorithm itself.

```
1  >        people_dict = {**self.actresses, **self.actors}
2  >
3  >        self.actresses = {*self.actresses.values()}
4  >        self.actors    = {*self.actors.values()}
5  >
6  >        for _ in range(self.m):
7  >            movie = Movie(data.pop())
8  >            for _ in range(int(data.pop())):
9  >                movie.add_cast(people_dict[data.pop()])
10 >            for person in movie.female_cast:
11 >                person.costarred |= movie.male_cast
12 >            self.actors -= movie.male_cast
13 >
14 >        self.full = True
15 >        if len(self.actors) > 0:
16 >            self.full = False
```

Note that a variable `self.full` is created to keep track of whether all actors played in a movie. The use for this will be explained in Section 3.3.3: Disconnected Actor.

This method of storing a set of `Actor` instances on each `Actress` instance proved more efficient in than creating a matrix representing edges, and more efficient than using a list of edges.

3.3 Matching

The matching algorithm went through several optimization steps. I'll introduce these optimizations gradually for simplicity. The most basic version of this algorithm is the following.

```
1    def pick_winner(self):
2        return "Mark" if self.match_all() else "Veronique"
3
4    def match(self, actress, i):
5        for actor in actress.costarred:
6            if actor.explored != i:
7                actor.explored = i
8                if actor.match is None or self.match(actor.match, i):
9                    actor.match = actress
10                   return True
11       return False
12
13   def match_all(self):
14       for i, actress in enumerate(self.actresses):
15           if not self.match(actress, i):
16               return False
17       return True
```

`pick_winner` is responsible for returning the winner. It simply extends `match_all` by changing the output from `True` or `False` to `Mark` and `Veronique`.
`match_all` is also simple. It iterates over all actresses, and calls `match` with each actress. This function returns `True` iff it succeeds in matching the given actress, and `False` otherwise. If `match` fails to match an actress, it cannot be matched. In such a case, it is impossible for a perfect matching to exist, and we know that Veronique wins. `match_all` returns `False` in this case.
If `match` returns true for all actresses, it means all actresses have been matched. Due to the 1-1 relationship between the number of actresses and actors, this means a perfect bipartite matching exists. Hence, `match_all` returns `True`.

`match` will iterate over all actors that have costarred with the actress v which was passed as a parameter. Each actor has an `explored` variable initialized to `-1`, which is used to track on which iteration the actor was last accessed. An actor can only be accessed again if it has not been accessed in this iteration before (line 6). If it is accessed, the `explored` variable is updated (line 7).
If some actor u is not matched with anyone, u will match with the actress v, and the function will return `True`.
If some actor u is matched with another actress w, the function will recurse using w. Because of the `explored` variable for each actor, the already accessed actor u is no longer eligible for matching on this iteration. This prevents infinite recursion of actress w wanting to match with actor u again.
If it is possible to find another match for actress w, then the match for actor u is updated to actress v. This recursive scheme ensures that if there is a possible way for some actress to be matched in the maximum matching of the graph, then it will be matched.
If the actress cannot be matched with any actor, the function returns `False`.

3.3.1 Ordered Costars

`match` is a function which may recurse in some cases. To optimize this function, we would like to reduce the amount of times this function needs to recurse. To do so, we realise that there are two outcomes causing the function to return `True`: the picked actor has no match, and a recursive case for the match of the picked actor.
We would like the first case to occur before the second case does. To accomplish this, we could sort the costars of the actress such that actors without a match occur first:

```
        ...
        for actor in sorted(actress.costarred):
        ...
```

with

```
    def __lt__(self, other):
        if not self.match:
            return True
        return False
```

added to the `Actor` class. This will ensure that actors without a match will be placed left, while actors with a match are placed on the right side of the sorted list.
This optimization improves the average running time between all samples recorded by Bram Pulles' `alg-tester`[1] from ∼555ms to ∼58ms.
However, sorting is slow. We can improve the performance even more by avoiding sorting alltogether, and simply duplicate the iteration over the costarring actors, where the first time we only consider non-matching actors, and vice versa for the second iteration.

```
    def match(self, actress, i):
        for actor in actress.costarred:
>           if not actor.match and actor.explored != i:
                actor.explored = i
                if actor.match is None or self.match(actor.match, i):
                    actor.match = actress
                    return True
>       for actor in actress.costarred:
>           if actor.match and actor.explored != i:
>               actor.explored = i
>               if actor.match is None or self.match(actor.match, i):
>                   actor.match = actress
>                   return True
        return False
```

This optimization improves the average running time recorded by the `alg-tester` from ∼555ms to ∼45ms. However, it breaks the programming principle of DRY (Don't Repeat Yourself), and has since been reduced to:

```
    def match(self, actress, i):
        for func in (lambda act: not act.match, lambda act: act.match):
            for actor in actress.costarred:
                if func(actor) and actor.explored != i:
                    actor.explored = i
                    if actor.match is None or self.match(actor.match, i):
                        actor.match = actress
                        return True
        return False
```

Unfortunately, function calls are relatively slow in Python. Because of this, the reduction increases the average recorded running time by some ∼3ms.

3.3.2 Ordered Actresses

By ordering the actresses based on the amount of people they have costarred with, least to most, we can reduce the amount of calls to `match` to 0.7871x as many. This corresponds to a reduction of 27.04%. The reason for this reduction is related to the previous optimization.

Actresses with just one costar have just one possible match. The actors these actresses match with cannot change their match, as it would leave the initial actress without a match.
These matches are confirmed at the start, as these actresses now get matched before ones with more costars. This causes the recently matched actors to be one of the last options for other actresses to try and match with. This is more efficient, as calls to try and match with these actors are wasted.

3.3.3 Disconnected Actor

If an actor or actress does not costar anyone, they cannot be matched by default. Hence, no perfect bipartite matching can exist, and `match_all` should return `False`. In Section 3.2.1: Input Parsing Optimization, the variable `self.full` serves the purpose of checking whether there are actors without a match. We can use this variable to immediately return `False`.

[1] An automated testing tool to test your algorithms, made by classmate Bram Pulles: https://github.com/Borroot/alg-tester

```
13      def match_all(self):
14          if not self.full:
15              return False
16
17          for i, actress in enumerate(sorted(self.actresses, key=lambda a: len(a.costarred))):
18              if not self.match(actress, i):
19                  return False
20          return True
```

Note that this variable only checks whether there is a disconnected actor. Section 3.3.2: Ordered Actresses is responsible for ensuring that if there is a disconnected actress, it will be matched first. In this case, no match will be found in $O(1)$ time, and the function returns `False` immediately.

3.3.4 Sparsely connected actors

As mentioned briefly in Section 3.3.2: Ordered Actresses, actors or actresses with just one costar must be matched with their costar in a perfect matching, if one exists. Because of this, additional preprocessing may be done to remove all of these actors and actresses from the graph. This would prevent other actresses from attempting to match with these "fixed" actors.
However, with my representation I was unable to find a method of doing this that would decrease the average running time, especially considering the optimization described in Section 3.2.1: Input Parsing Optimization prevents actors from having a set of costars.

3.3.5 Other Optimizations

- Each class uses a `__slots__` list. This list reserves space for the declared attributes and prevents the automatic creation of `__dict__` and `__weakref__` for each instance. Using it may speed up accesses of these attributes.
- The `Person` class has custom implementations of `__hash__` and `__eq__` to marginally improve performance of entering instances of the `Person` class in sets, and unioning those sets.

4 Complexity

4.1 Time Complexity

As mentioned previously, the program is split into two sections: an input parsing section and a matching section.

4.1.1 Input Parsing Time Complexity

Each line in the input parsing section is prefixed with the complexity for that line.

```
1             def parse_input(self) -> None:
2+4n+(2m*(2+s)) data = sys.stdin.read().splitlines()[::-1]
1               n, m = data.pop().split()
1               self.n = int(n)
1               self.m = int(m)
6
1               self.actresses = {}
n               for i in range(self.n):
n                   name = data.pop()
n                   self.actresses[name] = Actress(name, i)
11
1               self.actors = {}
n               for i in range(self.n):
n                   name = data.pop()
n                   self.actors[name] = Actor(name, i)
16
2n              people_dict = {**self.actresses, **self.actors}
18
n               self.actresses = {*self.actresses.values()}
n               self.actors    = {*self.actors.values()}
21
m               for _ in range(self.m):
m                   movie = Movie(data.pop())
m                   s = int(data.pop())
m*s                 for _ in range(s):
m*s                     movie.add_cast(people_dict[data.pop()])
m*fs                for person in movie.female_cast:
m*fs*ms                 person.costarred |= movie.male_cast
m*n                 self.actors -= movie.male_cast
30
1               if len(self.actors) > 0:
1                   self.full = False
```

where m and n are given, and s is the average cast size between all movies and fs and ms are the average female and male cast sizes between all movies respectively.
The complexity for line 2 is complex as it is the amount of lines in the input which are split up and reverted using `[::-1]`. The complexity from line 28 and 29 are based on the complexity for set operations in Python.
When adding these together and applying some simplifications, the complexity is the following:

$$9 + 12n + (2m * (2 + s)) + 2m + m * s + m * fs * ms + m * n$$

which can be simplified to

$$n + m * (s + fs * ms + n)$$

Hence, the average case time complexity for the input parsing phase is $O(n + m * (s + fs * ms + n))$.
The best case is when there are either no males, or no females in any of the movies. The complexity becomes $O(n + m * (s + n))$ then. The worst case complexity is when $fs * ms$ is maximal, which is when $fs = ms$, and there are as many females as males per movie.

4.1.2 Matching time complexity

```
    def pick_winner(self):
1       return "Mark" if self.match_all() else "Veronique"

    def match(self, actress, i):
2       for func in (lambda act: not act.match, lambda act: act.match):
2n          for actor in actress.costarred:
2n              if func(actor) and actor.explored != i:
n                   actor.explored = i
?                   if actor.match is None or self.match(actor.match, i):
n                       actor.match = actress
1                       return True
1       return False

    def match_all(self):
1       if not self.full:
1           return False

n log n for i, actress in enumerate(sorted(self.actresses, key=lambda a: len(a.costarred))):
n           if not self.match(actress, i):
1               return False
1       return True
```

where m and n are given, and s is the average cast size between all movies and ms is the average male cast size between all movies.
For this is the worst case scenario, each actress has all n actors as costars. Because of the outer for loop with 2 options, line 7 is called $2n$ times. But, due to how `func` partitions the actors, line 8 can be reached at most n times. Moreso, each time `match` is called from line 19, from inside `match_all`, each actor can only be accessed once. So, line 8 can only be accessed n times, which means recursion can only occur at most n times. So, lines 5 to 7 can be run at most n times. As this section has a complexity of $2 * n$ associated with it, the worst and average case time complexity for `match` is $O(2 * n * n + 1) = O(n^2)$.
The best case for `match` is when the actress has no costars. In this case we are dealing with $O(1)$ time complexity.

In the average and worst case, `match_all` will call `match` n times. Added together and slightly simplified, the result is:

$$2 + n \log n + n * n^2$$

Asymptotically, this reduces to $O(n^3)$ time complexity in the worst and average cases.
In the best case one of the actors is disconnected, and the optimization from 3.3.3: Disconnected Actor ensures that `self.full` is `True`, giving a time complexity of $O(1)$.

Due to how `pick_winner` is simply a wrapper for the `match_all` function, it copies all time complexities from it.

4.1.3 Total Time Complexity

The time complexity of the algorithm as a whole is simply the complexity from the input parsing section and the `pick_winner` function added together.
This gives a time complexity of $O(n + m * (s + fs * ms + n) + n^3) = O(m * (s + fs * ms + n) + n^3)$.

4.2 Space Complexity

As the matching section of the program does not use any space whatsoever, we will disregard it.

4.2.1 Input Parsing Space Complexity

On the left of the code, the total amount of space used relative to the input size will be tracked, at any specific line, in the worst case scenario. The worst case scenario is where $s = 2n$ for all movies. In short, everyone plays in all movies.

```
                    def parse_input(self) -> None:
1+2n+(m*(2+s))          data = sys.stdin.read().splitlines()[::-1]
2+2n+(m*(2+s))          n, m = data.pop().split()
3+2n+(m*(2+s))          self.n = int(n)
4+2n+(m*(2+s))          self.m = int(m)

4+2n+(m*(2+s))          self.actresses = {}
4+2n+(m*(2+s))          for i in range(self.n):
4+2n+(m*(2+s))              name = data.pop()
```

```
10 4+2n+(m*(2+s))                self.actresses[name] = Actress(name, i)
11 4+n+(m*(2+s))+3n
12 4+n+(m*(2+s))+3n          self.actors = {}
13 4+n+(m*(2+s))+3n          for i in range(self.n):
14 4+n+(m*(2+s))+3n              name = data.pop()
15 4+n+(m*(2+s))+3n              self.actors[name] = Actor(name, i)
16 4+(m*(2+s))+8n
17 4+(m*(2+s))+16n           people_dict = {**self.actresses, **self.actors}
18
19 4+(m*(2+s))+16n           self.actresses = {*self.actresses.values()}
20 4+(m*(2+s))+16n           self.actors    = {*self.actors.values()}
21
22 4+(m*(2+s))+16n           for _ in range(self.m):
23 5+(m*(1+s))+16n               movie = Movie(data.pop())
24 6+(m*s)+16n                   s = int(data.pop())
25 6+(m*s)+16n                   for _ in range(s):
26 6+(m*s)+16n                       movie.add_cast(people_dict[data.pop()])
27 6+(m*s)+16                    for person in movie.female_cast:
28 6+(m*s)+16n+n*n                   person.costarred |= movie.male_cast
29 6+(m*s)+13n+n*n               self.actors -= movie.male_cast
30 4+13n+n*n
31 4+13n+n*n                  if len(self.actors) > 0:
32 4+13n+n*n                      self.full = False
```

After `parse_input` returns, all local variables are removed. This leaves $2 + 8n + n * n$. This consists of

- 2 from n and m.
- $5n$ from storing 5 attributes per n actors.
- $3n + n * n = n(n + 3)$ from storing 3 attributes and n actors per n actresses.

Asymptotically, this reduces to a worst case space complexity after parsing of $O(n^2)$. This complexity also applies for the average case.
However, the best case scenario is just $O(n)$. In this scenario only actors play in movies, which means no actresses will have to store references to actors.
It's worth noting that the function may have a higher memory use at the peak, around line 28, than it does after returning.

4.3 Local Running Time

Sample	Parsing	Matching	Total	Ratio
a1.in	0.0002	0.00001	0.00021	18.68
a2.in	0.0001	0.00001	0.00008	13.42
m1.in	0.0002	0.00001	0.00020	12.82
m2.in	0.0001	0.00001	0.00013	8.33
m3.in	0.0002	0.00002	0.00024	8.99
m4.in	0.0001	0.00001	0.00014	11.40
v1.in	0.0002	0.00002	0.00018	6.21
v2.in	0.0006	0.00010	0.00066	5.41
v3.in	0.0003	0.00003	0.00031	8.45
v4.in	0.0004	0.00003	0.00041	11.70
r1.in	0.0182	0.00000	0.01819	6936.09
r2.in	0.0163	0.00028	0.01662	59.36
r3.in	0.0188	0.00661	0.02538	2.84
r4.in	0.0305	0.00945	0.03997	3.23
r5.in	0.0399	0.02852	0.06840	1.40
r6.in	0.0338	0.02498	0.05880	1.35
r7.in	0.0299	0.00000	0.02993	12551.20
r8.in	0.0330	0.00022	0.03323	151.47
r9.in	0.0011	0.00017	0.00131	6.49
r10.in	0.0012	0.00000	0.00121	1014.20

The parsing, matching and total column values are all in seconds.
The matching for the `a*.in`, `m*.in` and `v*.in` samples take a very insignificant amount of time for parsing, and even more so for matching.
The ratio between parsing and matching is still high, but this is explained due to the low amount of recursive calls required to match all actresses in these cases. Performance of the matching of these simple samples are closer towards $O(n)$ than $O(n^3)$.

For the `r*.in` samples, the results get a bit more extreme. Notable samples are `r1.in`, `r7.in` and `r10.in`, for which the running time of the matching phase is incredibly low due to the optimization described in 3.3.3. For these cases the parsing time is orders of magnitudes larger than the actual algorithm time.

5 Acknowledgements

I would like to mention that my classmate Bram Pulles' `alg-tester`[2] has been an invaluable resource in testing and especially in optimizing both my algorithm for this project, as well as the one from the previous project. It allows a user to test their algorithm using any samples, while reporting running times and accuracy, regardless of what language the algorithm was written in. He selflessly put a lot of work into the project, and believe he deserves some recognition for his work.

[2] https://github.com/Borroot/alg-tester

6 Appendix

6.1 Problem

In addition to the aforementioned problem, a secondary problem was presented.
or this part you will play the above game against other teams. Of course, since you already know the winning strategy, doing exactly this would be boring. So we add the following scoring mechanism:
A move $x_i \to y_i$ will be awarded k points, where k is the number of movies where x_i and y_i both play in. (Similarly for a move $y_i \to x_{i+1}$.) If you lose in the end, you will get 0 points. If you win, you get average score of your moves. (If you win without making any moves, you'll get the highest possible k as score.)
Your program will run against every other program twice; once where you make the initial move (i.e. you are Veronique), and once where you make the second move (i.e. you are Mark). Your score will be summed up among all matches.

6.2 Strategy

There are exactly four possible scenarios. For each of the scenarios, a different strategy applies.

1. Mark wins, we are Mark.
2. Veronique wins, we are Mark.
3. Mark wins, we are Veronique.
4. Veronique wins, we are Veronique.

6.2.1 Mark wins, we are Mark

We apply exactly Mark's strategy proposed in Proof 2.2. Assuming Veronique previously said actress x, we will respond with the match of x. As shown in Proof 2.2, this strategy is guaranteed to result in a win.
Note that this strategy is deterministic. There is always only one option for us. Hence, we do not attempt to get a higher k at the risk of losing.

6.2.2 Veronique wins, we are Mark

Three possible strategies exist for this scenario:

1. We try to finagle a win, by picking unexpected actors in an attempt to force a mistake by Veronique.
2. We accept the loss, and try to ensure the lowest k for Veronique.
3. We simply respond with pickable actors arbitrarily, hoping for a mistake by Veronique.

Due to performance reasons related to finding the best k, we have opted for the last option.

6.2.3 Mark wins, we are Veronique

Just like in the previous section 6.2.2, three options apply.

1. We try to finagle a win, by picking unexpected actresses in an attempt to force a mistake by Mark.
2. We accept the loss, and try to ensure the lowest k for Mark.
3. We simply respond with pickable actresses arbitrarily, hoping for a mistake by Mark.

Again, we choose the last option for performance reasons.

6.2.4 Veronique wins, we are Veronique

We apply exactly Veronique's strategy proposed in Proof 2.2. This means that we always pick an unmatched actress. We have already shown that if Veronique wins, she can always pick an unmatched actress. This strategy is guaranteed to result in a win.
However, unlike in section 6.2.1, this strategy is not deterministic. It is possible for multiple unmatched actresses to be pickable. However, due to performance reasons, we have opted not to apply any strategy on which actress would be picked, as we fear the program would run for longer than is allowed (30 seconds).
So, we respond with the first unmatched actress adjacent to the previously picked actor. On the first turn, we simply pick the first unmatched actress.

6.3 Overall Strategy

The overall strategy is twofold:

- If we are on the winning side:
  - Guarantee the win. Take no risks.
- If we are on the losing side:
  - Pick allowed responses arbitrarily, in an attempt to not time out.