El equipo ideal

Materia: Programación 3
Comisión: 02
Alumnos: Hernan Borja, Echabarri Alan
Profesores: Daniel Bertaccini, Gabriel Carrillo

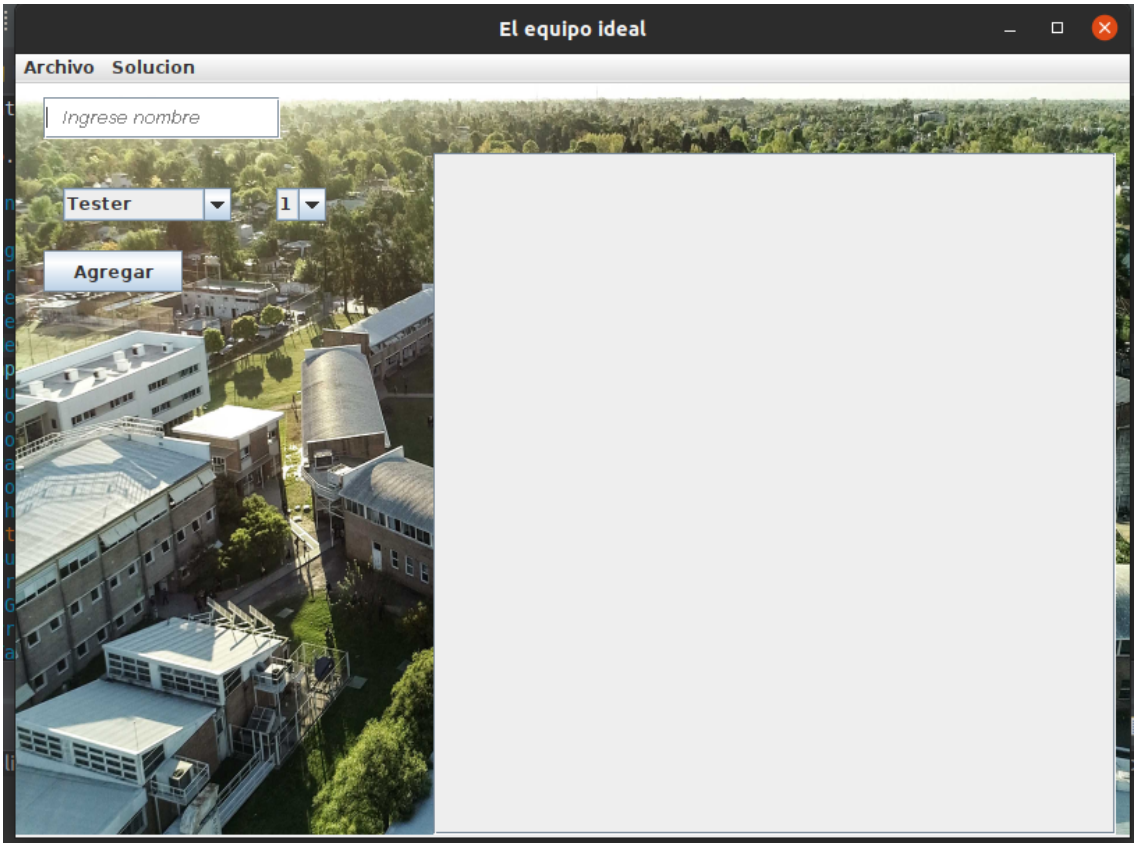
Índice

Como usar el programa

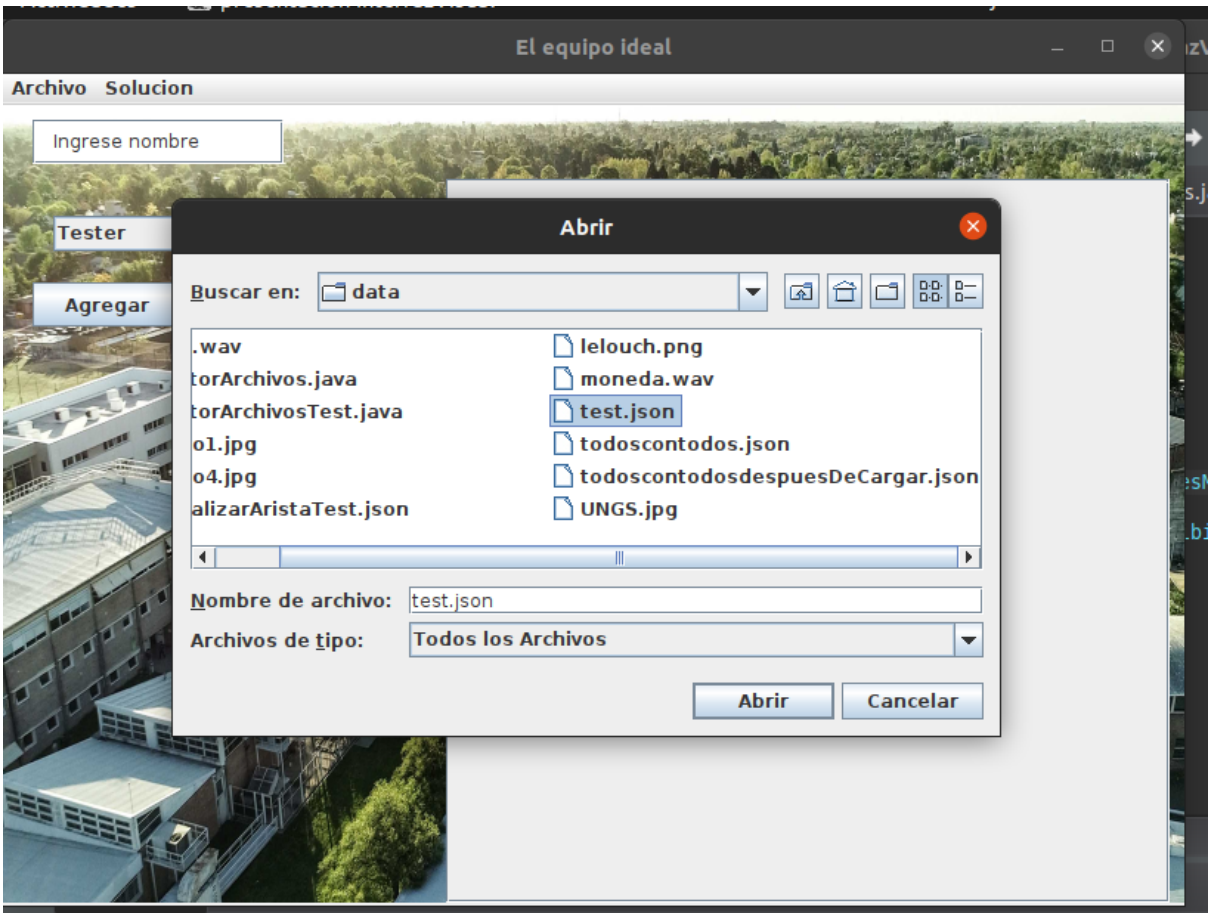
Para ejecutar el programa siga los siguientes pasos:

1. git clone https://github.com/alan1996colo/elEquipoIdeal.git
2. drijase a *elEquipoIdeal/src/presentacion*
3. con eclipse o su ide favorito abra y ejecute el archivo *InterfazVisual.java*

   Deberia ver algo asi.



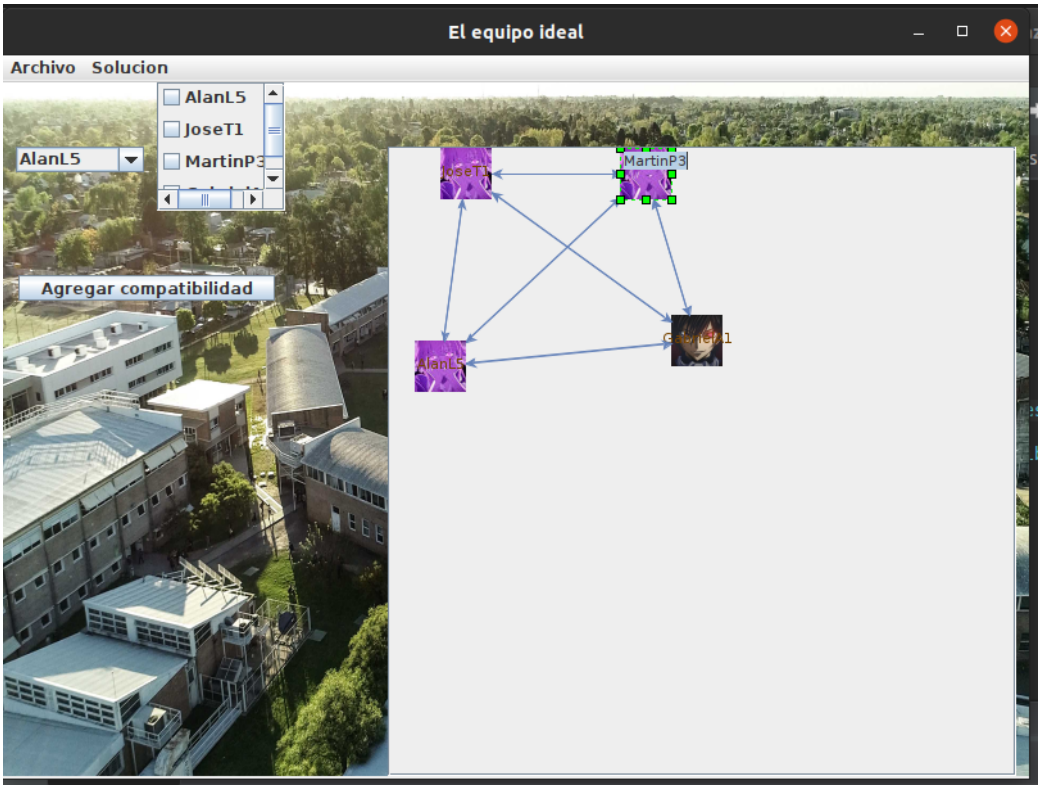
Puede ir agregando personas manualmente a la “empresa” o puede cargar una lista de personas que anteriormente ya haya guardado desde la pestaña archivo cargar, asegúrese de seleccionar un archivo con extension .json



sugerimos que use su propia lista de empleados previamente guardada.

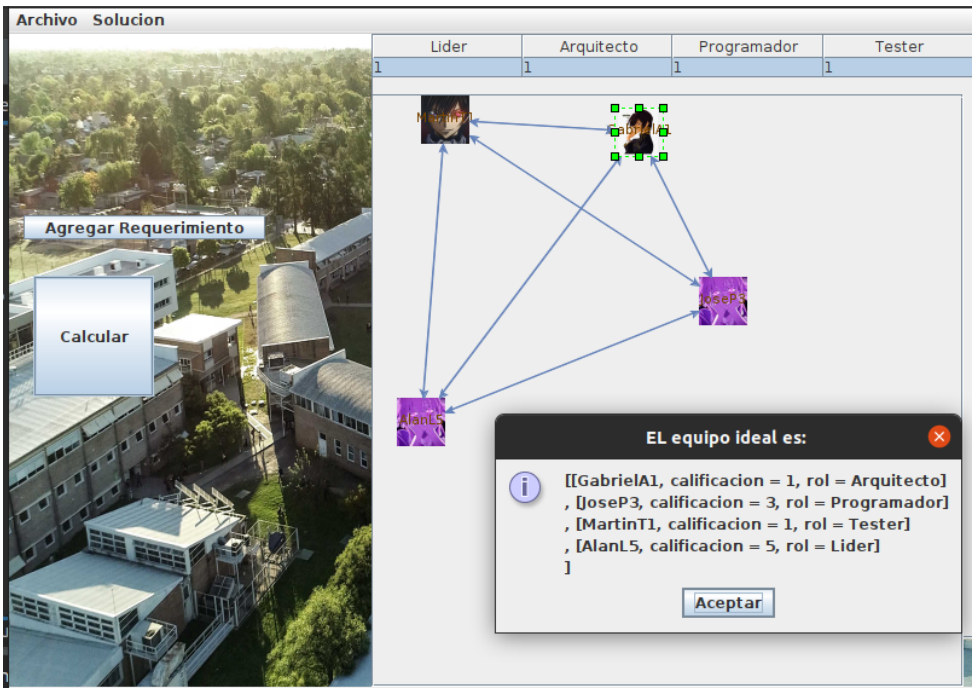
Atención! *Sugerimos que al ir agregando personas a la empresa use la siguiente nomenclatura como un recurso para la memoria. NombreLetraRolCalificacion. Ejemplo, queremos ingresar a la persona Joaquin Lider calificación 5. → JoaquinL5.*

Una vez haya terminado de ingresar la lista de empleados debe proceder a agregar sus compatibles. Vaya a la pestaña solución →crear lista de compatibles.



Y vaya marcando y presionando sobre agregar compatibilidad.

Luego dirijase a la pestaña solución→cargar modificar/requerimientos. ingrese los requerimientos necesarios y presione agregar, luego calcular.
Finalmente debe visualizar algo como esto.



Decisiones de implementación

En este trabajo hubo varias decisiones de implementación importantes.

- La primera respecto del tipo de representación del grafo. Optamos por la implementación en lista de vecinos (adyacencia) por su orden de complejidad. Además resultaba más sencillo agregar y leer los datos de entrada de cada arista.
- Dividimos en capa de negocio y la interfaz visual tal como era el requerimiento.De manera que la capa de Presentación llama a la capa de negocio, pero la capa de negocio desconoce a la capa de presentación.

Desarrollo

interpretación del problema

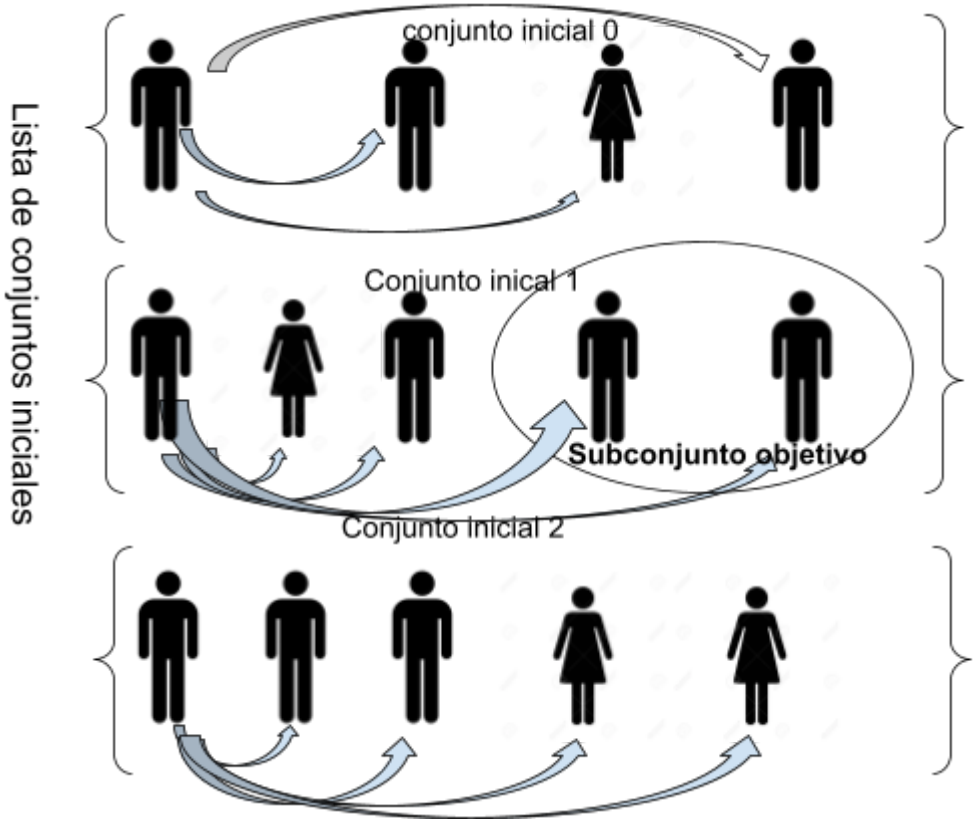
Para abordar el problema principal del enunciado, lo primero que se nos ocurrió fue que podíamos reciclar el codigo de negocio del trabajo anterior.
Por lo tanto desde nuestra interpretación, cada persona de la empresa seria un vértice del grafo, y la “compatibilidad” serian aristas. Siendo que si una persona es compatible con otra, quiere decir que existe una arista entre los vértices que las representan.

Luego agregarle a esos vértices los atributos “calificación” y “rol” , y algunos métodos pertinentes.

Desarrollo del algoritmo principal.

Para implementar el algoritmo principal, usamos como lógica inicial la siguiente:

“Si existe un subconjunto tal que cumpla:
1.Son todos compatibles entre si.
2.El subconjunto cumple los requerimientos solicitados.
3. Es el subconjunto mayor calificado.



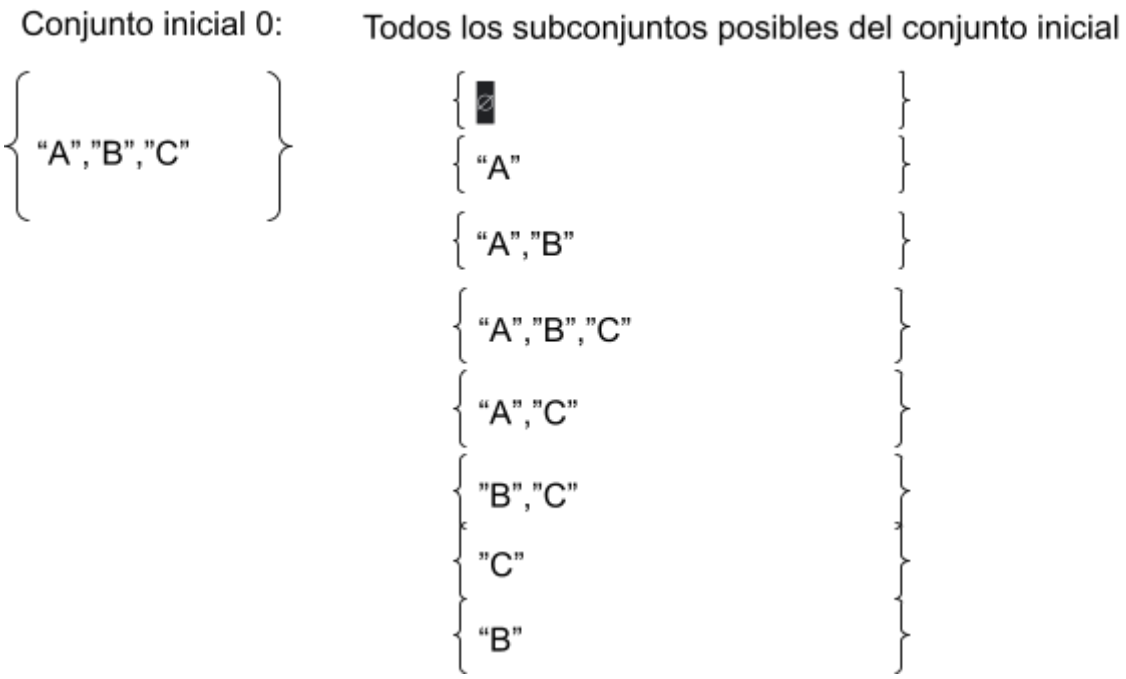
<<Las flechas muestran muestran como la primera persona del conjunto es siempre compatible con sus compatibles, resto de personas del conjunto>>

Entonces, ese subconjunto debe encontrarse dentro de alguno de los conjuntos iniciales. Ya que por como implementamos el grafo, cada vértice solo contiene aristas y vértices compatibles.
Y globalmente ese vértice , junto con sus vertices compatibles, es un conjunto inicial.
Por lo tanto, para reducir la entrada para algún futuro algoritmo de fuerza bruta, aplicamos un filtrado quitándonos los conjuntos que no satisfacen los requerimientos minimos.

Es decir si los requerimientos son 1 tester.
Y uno de los conjuntos iniciales (conjuntos grandes) no puede satisfacer ese requerimiento, es decir, no puede tener 1 o mas tester.
quiere decir que ese conjunto no tiene una solución.

Lo siguiente fue usar el algoritmo de backtracking visto en clase, con algunas modificaciones mínimas para resolver el problema.

Recorremos la lista de conjuntos iniciales ya filtrada.



para cada conjunto inicial de la lista, creamos todos los subconjuntos posibles.
en cada uno de todos los subconjuntos posibles, chequeamos que cumplan los requerimientos estrictamente. (en este caso 1==1), y que sean todos compatibles con todos(dentro del subconjunto actual iterado), si se satisface esto, comparamos con el ultimo mejor subconjunto almacenado anteriormente, si es mejor, reemplazamos y seguimos. Si es peor, no reemplazamos y seguimos.
Finalmente devolvemos el subconjunto mejor candidato.
Claramente este algoritmo es una fuerza bruta, que tiene una complejidad exponencial O(2**n) donde n es la entrada lista de conjuntos.

Desarrollo del algoritmo heurístico.

Para el desarrollo de la heurística , decidimos usar el algoritmo principal, pero no mirar todos los casos posibles, es decir hacer solo una búsqueda local.

Por lo tanto el código de la heurística es prácticamente el mismo que el principal pero sin generar todos los subconjuntos posibles.
por lo tanto la heurística es orden polinomial.

- ventajas: menor complejidad, se resuelve en menor tiempo.
- Desventajas: podría no encontrar solución, ni la solución óptima ya que solo se queda con una búsqueda local y no revisa todos los subconjuntos posibles. (Podría darse el caso que la unica solución se encuentre en un subconjunto, dentro de un conjunto inicial, pero que el orden de los elementos no favorezca al algoritmo, por lo tanto no encuentre esa solución.)

Desarrollo de hilos

Para la implementación de hilos del algoritmo de fuerza bruta, se hicieron algunas modificaciones en la clase que contiene al codigo principal de fuerza bruta. Esta clase se llama Solver, y inicialmente solo recibia 2 parametros , pero para implementar hilos debimos agregarle 2 parametros adicionales para manejar concurrencia. Esos parámetros son un Objeto de bloqueo o lock, y la variable a la que deben acceder todos los hilos para comparar su resultado.

Luego desde la clase controladora *Negocio* , en el metodo que llama a los hilos le agregamos una utilidad para que no retorne un resultado hasta que todos los hilos terminen.

TAD

Paquete Negocio:

Clase Negocio:

La clase “negocio”, será la encargada de realizar la lógica del programa, esta contenera al grafo a utilizar, la cual serán cargadas con los datos que ingrese el usuario

Atributos:

-GrafoLista personas: Contienen a todas las personas
-Requerimiento req: Contiene los requerimientos , las cantidad de personas solicitadas.

Metodos:

-void AgregarPersonas(..): Agrega a una persona con , rol , apellido y calificacion
-void CargarRequerimientos(..): Carga los requerimientos, si existe algún valor negativo o la cantidad es superior a los disponibles lanza excepciones.
-void cambiarSesion(..): Reemplaza el grafo actual por el cargado desde el archivo y los inicializa.
-void guardarSesion(..): Crea un archivo y guarda a la personas ingresadas.
-boolean agregarCompatibilidad(..): Dado dos apellidos pasados , lo busca en el grafo y agrega la arista.
String getNombres(): Retorna todos los nombres ingresados.
ArrayList<String> getCompatiblesDe(String nombre): Busca la persona en el grafo por el nombre, cuando la encuentra devuelve una lista de sus compatibles.
-boolean agregarInCompatibles(..): Agrega a las personas incompatibles de una persona.
-**calcularEquipoIdeal**(..): calcula el equipo ideal usando el atributo grafo y requerimiento, se ejecuta en tantos hilos como vetices, y cuando todos los hilos hayan terminado devuelve un string con la lista de personas contenidas en el equipo ideal.

Clase GrafoLista:

La clase “GrafoLista” contendrá a todas las personas de la empresa, cada persona contenera una lista de vecinos, se podra ingresar a las personas , sus aristas correspondientes(sus compatibles)

Atributos:

List<Persona>Personas : Contiene todas las personas.

Metodos:

-GrafoLista(..) : Constructor.
-void setPersonas(..): Cambia la lista de personas a las lista pasada.
-Persona getPersonaNum(..): Devuelve una persona segun el indice pasado.
-void AgregarPersona(..): Recibe una Persona y la agrega, si ya la contiene tira excepción, si no la agrega.
-void AgragarArista(..): Agrega la arista entre dos personas compatibles.
-List<Persona>getPersonas(): Devuelve las personas del grafo.
-int getTamanio(): Devuelve el tamaño dell grafo.
-Set<Arista> getVecinos(..): Devuelve los vecinos según el índice ingresado.
-boolean contains(..): Verifica si una persona esta en la lista de personas.
-void mostrarGrafo(): Muestra el contenido de grafo.
-boolean sonVecinos(..): Devuelve verdadero si dos personas son vecinos, false si no lo son.
-void inicializarAristas(): Inicializa todas las aristas del grafo.
-Persona getPersona(..): Retorna el objeto Persona, devuelve el primero que encuentre.

Clase Persona:

La clase “Persona” contiene los atributos de una persona , apellido , calificación, rol y una lista de aristas en la cual serian las personas compatibles.

Atributos:

-String apellido: Apellido de la persona.
-int calificacion : Calificación de la persona , va de 1 a 5.
-string rol: Profesión de la persona , puede ser arquitecto, tester, programador o lider.
-HashSet<Arista>compatibles: conjunto de compatibles con la persona.

Metodos:

-Persona(..): Constructor
-Set<Aristas> getCompatibles(): Devuelvo los vecinos de un nodo.
-boolean estaPersona(..): Dado una persona ingresada, devuelve true si esta en el conjunto de aristas y false si no.
-void agregarCompatibles(..): Dado una persona destino , verifica primero si la persona existe, si existe genera excepción si no agrega a compatibles la persona.
-String getApellido(),void setApellido(),int gerCalificaion(),void setCalificacion(),string getRol(): Son los getters y setters de los atributos.
-boolean quitarCompatibles(..): Dada una persona destino saca del conjunto de arista a la persona.

Clase Arista:

La clase “Arista” contendrá a dos personas, persona origen y destino, si esta dos personas están significa que son compatibles.

Atributos:

-Persona PersonaOrigen: Es la persona origen.
-Persona PersonaDestino : Es la persona destino.
-String personaOrigenNombre: Contiene el nombre de la persona origen.
-String personaDestinoNombre: Contiene el nombre de la persona destino.

Metodos:

-Arista(..): Constructor.
-getPersonaOrigen(),getPersonaDestino(),getPersonaOrigenNombre(),getPersonaDestinoNombre(): Son los getters y setters de los atributos.

Clase Requerimiento:

La clase “Requerimiento”, tendrá la cantidad y el rol de personal solicitada por el usuario.

Atributos:

-int cantLiderProyecto: Cantidad de Líderes solicitado.
-int cantArquitecto: Cantidad de Arquitectos solicitado.
-int cantProgramadores: Cantidad de Programadores solicitado.

-int cantTester: Cantidad de Tester solicitado.

Metodos:

-Requerimiento(..): Constructor.
-boolean cumpleRequerimientos(..): Dado la cantidad solicitada de roles , verifica si las cantidades son correctas.
-boolean superaRequerimientos(..): Dado la cantidad solicitada de roles , verifica si la cantidad solicitada supera a la cantidad establecida, si es asi retorna false , de lo contrario true.
-int getCantLiderProyecto(),void setCantLiderProyecto(),int -getCantArquitecto(),void setCantArquitecto(..),int getCantProgramador(),void setCantProgramador(..),int getCantTester(), void setCantTester(..) : Son los Setters y Getters de los atributos.

Paquete Compatibles:

Clase solver:

La clase “solver”, contenera el algoritmo de fuerza bruta para realizar la búsqueda del equipo ideal

Atributos:

GrafoLista grafo: Contendra el grafo a resolver
HashSet<Persona> _mejor: aqui se ira sobreescribiendo el mejor conjunto.
Requerimiento _req; son los requerimientos solicitados por el cliente.
Object bloqueo; se usa para manejar concurrencia con hilos.

Metodos:

-Solver(..): Constructor , le pusimos sobrecarga para poder usar con o sin hilos a gusto.

-calcular(...): retorna el conjunto o subconjunto mayor calificado que cumple los requerimientos y en el cual todos sus elementos sean compatibles con todos sus elementos.
-filtrarLosQueSuperanRequerimiento(...): modifica la lista de conjuntos ingresada, para dejar solo aquellos conjuntos que superan requerimientos.
-esMejorCalificadoQueElSet(...): Revisa si el primer conjunto ingresado por parametro es mejor calificado que el segundo conjunto ingresado por parametro, y devuelve verdadero si esto es cierto.

-transformarPersonaToSet(...): Dado un vertice/persona ingresado, retorna un conjunto en el cual se encuentra ese vertice y todos sus compatibles.
-conjuntoCumpleRequerimientos(...): revisa estrictamente que el conjunto ingresado cumpla los requerimientos ingresados. si el conjunto supera los requerimientos o el conjunto esta por debajo de los requerimientos devolvera false.
-conjuntoSuperaRequerimientos(...):revisa que el conjunto supere o sea igual a los requerimientos ingresados, de lo contrario devolvera false.
-obtenerTodosLosSubConjuntosPosibles(...): dado el conjunto ingresado, devuelve una lista con todos los subconjuntos posibles del conjunto ingresado.
-obtenerSubconjuntosAux(...): metodo auxiliar de obtenerTodosLosSubConjuntosPosibles, aqui ocurre el bactracking y la recursividad.
-resolverHeuristica(...): busca una solucion heuristica o con algoritmos heuristicos, para encontrar el conjunto que cumpla los requerimientos, sea mejor calificado y sean todos compatibles. Puede no encontrar solucion optima, como no encontrar ninguna solucion.
-isTheBestSet(...): retorna verdadero si el conjunto ingresado, tiene la mejor calificacion posible para los requerimientos solicitados, sino falso.
-reducirConjunto(...):quita las personas sobrantes de un conjunto para quedarse con un conjunto que cumpla estrictamente el requerimiento.
-run(..): se usa para hilos, ejecuta el metodo calcular y si su resultado es un conjunto mejor que el conjunto guardado en el atributo mejor, entonces guarda su resultado alli.

Clase Auxiliares:

La clase Auxiliares solo se usa con un unico fin que es comprobar si en un conjunto, todos sus vertices son compatibles entre si.

-**sonCompatibles(...):** retorna falso si existe al menos un vertice que no sea compatible con el resto(siempre y cuando sean vertices distintos.)

Dificultades encontradas

serializacion/deserializacion:
intentamos muchas formas de que al serializar el grafo se guarden las aristas/compatibles, el primer problema encontrado es que en nuestras aristas nosotros guardamos referencia al Objeto Persona, creimos resolverlo al indicarle a java que los objetos Persona dentro del Objeto Arista , eran No serializables, y para compensarlo, crear un Objeto String personaNombre, de manera que no tenga problemas de recursividad al guardar el archivo preety json.
Por un momento creimos que todo marchaba bien, el archivo se guardaba y se cargaba correctamente, pero luego encontramos otro error de comportamiento.
Al cargar el archivo serializado a nuestro grafo, nos encontrabamos con que visualmente, las aristas no se mostraban en la interfazGrafica. Realizamos algunos test, y revisamos el archivo json con un bloc de notas, y los test salian fallidos, pero desde el archivo json en el bloc de notas podiamos ver que si se serializaba correctamente, entonces concluimos que era un error de carga. Inicialmente perdimos mucho tiempo pensando que era algun error en la implementacion logica del metodo inicializarAristas() del clase grafo, que dado un grafo, revisa todos los vertices, mira sus compatibles, y con el String nombre, comienza a agregar nuevamente los Objetos Persona como referencia que se habian quitado para poder serializar.

Despues de perder mucho tiempo con un algoritmo, que deberia no habernos causado mas problemas que algun error de concurrencia por no usar Iterator(), nos decidimos a crear nuevos test. En esos test pudimos notar que el tamaño del hashSet compatibles ,luego de cargarlo al grafo, es decir deserializarlo, era menor al tamaño del hashSet inicialmente guardado. Por lo tanto concluimos que el problema no estaba en inicializarArista(), sino en cargarListaDesdeJson().
Finalmente por falta de tiempo , decidimos no continuar con la resolucion del metodo cargarListaDesdeJson(), este metodo

actualmente te cargara el grafo, pero solo con una arista en cada vertice.
Tambien se perdió mucho tiempo en un error lógico en el desarrollo del algoritmo de fuerza bruta, inicialmente se hizo que el algoritmo principal filtre a los conjuntos que no cumplan estrictamente los requerimientos antes de crear los subconjuntos. Luego de agregar bastantes test, descubrimos que el error era ese mismo, y se cambio el filtrado inicial a que superen los requerimientos.

Tareas pendientes

Nos quedaron pendientes algunas mejoras en la interfaz visual para mejorar la experiencia de usuario.

- ☐ Aplicar ley de morgan: en lugar de mostar carteles de excepciones, hubiera sido mejor no mostrar algunos componentes, como por ejemplo cuando se selecciona compatibilidad con quitar la opción de elegir a la misma persona como destino.
- ☐ Agregar funcionalidades adicionales, como cargar tu propia imagen al vértice, o reproducir sonidos cuando se presiona algun boton.
- ☐ Arreglar la asignación de vertices dinámica a la pantalla; no se trabajo mucho eso y por ello es que se va reseteando la posición cada vez que se cambia de pantalla.
- ☐ Mostrar en el vertice el rol y calificacion ademas del nombre de la persona, esto seria bastante util para darse una idea a simple vista de si el resultado es correcto.
- ☐ Cambiar el mensaje de resultado por una nueva pantalla emergente con un mini grafo.

Desde la parte de buenas practicas de programacion y clean code, quizás algunas de las mencionadas a continuacion no se ven completamente reflejadas

- ☐ Refactorizar lo mayor posible al punto de un nivel de lenguaje natural.
- ☐ Renombrar variables y métodos por los nombres mas descriptivos posibles.
- ☐ Hacer el codigo lo mas general posible para que en un futuro pueda ser más versátil de adaptarse a otros requerimientos.

Librerías utilizadas

Para este trabajo se utilizaron las siguientes librerías para la interfaz Gráfica.

1. JGraphX:
    - Nombre de la librería: JGraphX
    - Sitio web oficial: https://jgraph.github.io/mxgraph/
    - Repositorio de GitHub: https://github.com/jgraph/mxgraph
    - Licencia: JGraphX utiliza la licencia dual GNU General Public License (GPL) v3 y la licencia comercial de JGraphX.
2. Gson:

    - Nombre de la librería: Gson (Google Gson)
    - Sitio web oficial: https://github.com/google/gson
    - Repositorio de GitHub: https://github.com/google/gson
    - Licencia: Gson se distribuye bajo la licencia Apache License 2.0.