Informe de Trabajo Final

Buscador de coincidencias

- Alumna: Romero Rocío Jazmín
- Profesor: Leandro Caballero.

- Materia: Complejidad Temporal, Estructuras de Datos y Algoritmos.
- Carrera: Ingeniería en Informática.
- Ciclo lectivo: 2° cuatrimestre del 2025.
- Fecha: 24/10/2025

Contenido

Introducción …… 03
Desarrollo ……..…… 04
Detalles de la implementación …… 05
    I. Imágenes del código ……..…… 06
    II. Interfaz del usuario final ……..…….. 13

Errores detectados en la ejecución del programa ……..…… 16
    I. Error en .NET ……16
    II. Errores en archivos .resx…….……..……16
    III. Otros errores encontrados ……..……. 17

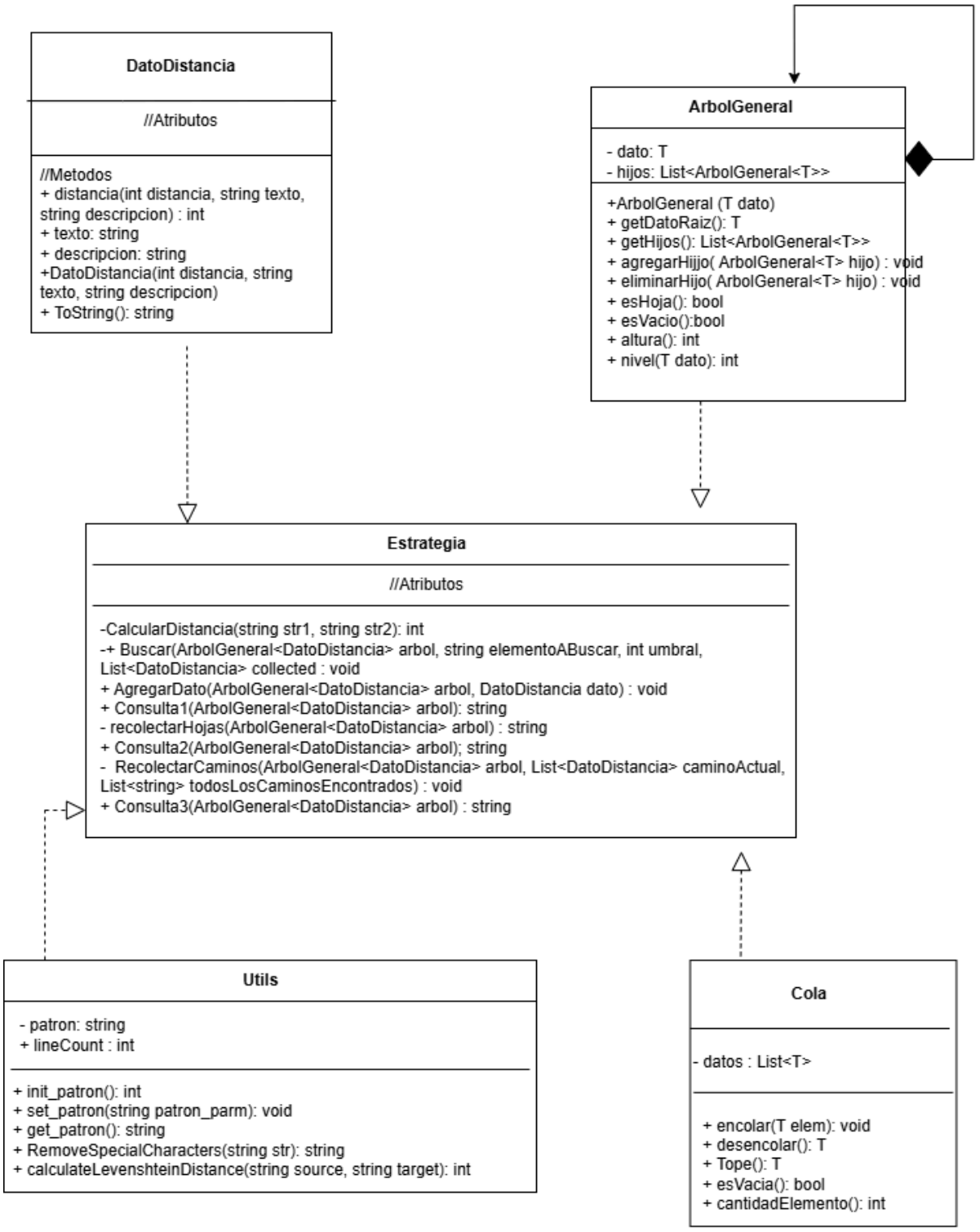
Problemas al realizar el código ... ……. 19
Posibles mejoras …..…… 20
Conclusión ……..…. 21

Introducción:

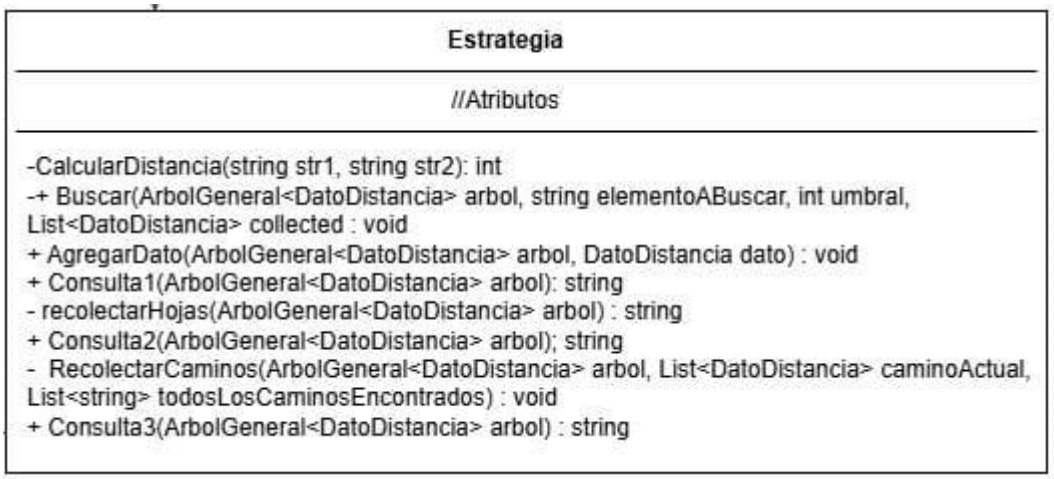
El objetivo principal del proyecto fue el desarrollo de un sistema de búsqueda de coincidencias aproximadas, utilizando como estructura de datos un árbol BK (Burkhard-Keller tree). Esta estructura está diseñada para la indexación y búsqueda eficiente de datos en espacios métricos, empleando en este caso la distancia de Levenshtein como función para medir la similitud entre cadenas de texto. La aplicación permite al usuario seleccionar un archivo de datos en formato CSV para indexarlo. A partir de esta base de datos de películas, el sistema construye un árbol BK que optimiza el proceso de consulta, permitiendo realizar búsquedas flexibles basadas en una coincidencia exacta que ingrese el usuario, en un umbral de similitud o "distancia" definido por el usuario.

Desarrollo:

El siguiente diagrama de clases UML describe la estructura del sistema.



Para observar de forma más clara la clase en la que hubo cambios:



Detalles de la implementación:

I. Imágenes del código

La clase Estrategia se encarga de implementar toda la lógica para construir, manipular y consultar un árbol BK (Burkhard-Keller tree). Esta estructura de datos, utilizando la distancia de Levenshtein como métrica. Los métodos que se implementaron para gestionar el árbol son:

• **AgregarDato**: Añade un nuevo elemento al árbol BK, calculando su distancia con el nodo actual y ubicándolo en la rama correspondiente para mantener la estructura ordenada del árbol.

• **Buscar:** Ejecuta una búsqueda por proximidad dentro del árbol, retornando una lista de todos los elementos que se encuentran dentro de un umbral de distancia específico respecto al término buscado.

• **Consulta1**: Recorre la estructura para identificar y devolver un listado con todos los nodos que son hojas del árbol.

• **Consulta2:** Genera un reporte que detalla todos los caminos posibles desde el nodo raíz hasta cada una de las hojas.

• **Consulta3:** Devuelve un texto con los datos del árbol organizados por el nivel en el que se encuentra cada nodo, ofreciendo una vista de la estructura por profundidad.

```
private int CalcularDistancia(string str1, string str2)
    {

      // Si cualquiera de las dos cadenas es nula o sólo espacios, devuelvo un valor grande (no coincidenci
      if (string.IsNullOrWhiteSpace(str1) || string.IsNullOrWhiteSpace(str2))
      {
          return 1000;
      }

      // Saco espacios alrededor y paso todo a minúsculas para comparar sin distinguir mayúsculas
      str1 = str1.Trim().ToLower();
      str2 = str2.Trim().ToLower();

      // Si las cadenas completas son exactamente iguales la distancia es 0
      if (str1 == str2)
      {
          return 0;
      }

      // divido cada cadena en palabras separadas por espacios, evitando entradas vacias
      String[] strlist1 = str1.Split(' ', StringSplitOptions.RemoveEmptyEntries);
      String[] strlist2 = str2.Split(' ', StringSplitOptions.RemoveEmptyEntries);


      //inicializo la distancia con un valor grande para poder tomar el minimo despues
         int distance = 1000;

      //recorro cada palabra del primer texto
         foreach (String s1 in strlist1)
         {
          //recorro cada palabra del segundo texto
             foreach (String s2 in strlist2)
             {
              //si alguna palabra coincide la distancia es inmediatamente 0
              if(s1 == s2)
              {
                  return 0;
              }

              //si no coincide exactamente, calculo la distancia de levenshtein entre las dos palabras
              // y me quedo con la minima encontrada hasta ahora
                 distance = Math.Min(distance, Utils.calculateLevenshteinDistance(s1, s2));
             }
         }

      //devuelvo la distancia minima encontrada, si no hubo cambios quedará en 1000 y no devuelve nada
         return distance;
    }
```

El método **CalcularDistancia** devuelve un entero si dos cadenas de texto son iguales. Primero valida que ambas sean válidas. Si alguna está vacía, retorna 1000 indicando que no hay coincidencia. Luego normaliza los textos convirtiéndolos a minúsculas y eliminando espacios. Si tras esta limpieza las cadenas son iguales, devuelve 0. Si no hay coincidencia exacta, divide cada cadena en palabras y

compara cada una de la primera con cada una de la segunda. Si encuentra dos palabras idénticas, retorna 0 de inmediato. En caso contrario, calcula la distancia de edición entre cada par de palabras mediante **Utils.calculateLevenshteinDistance**, conservando el valor mínimo obtenido. Finalmente, devuelve esa distancia mínima, que indica el grado de similitud entre ambos textos (cuanto menor es el valor, mayor es la semejanza)

```
//metodo que busca en el arbol todos los nodos cuya distancia al texto buscado sea <= umbral
public void Buscar(ArbolGeneral<DatoDistancia> arbol, string elementoABuscar, int umbral, List<DatoDistancia> collected)
{
    // Si el árbol o la lista de resultados es nula, no hago nada y retorno
    if (arbol == null || collected == null)
    {
        return;
    }

    // Si el texto a buscar es nulo, lo convierto en cadena vacía para evitar excepciones
    if (elementoABuscar == null)
    {
        elementoABuscar = "";
    }

    // Calculo la distancia entre el texto nodo actual y el buscado
    int distancia = CalcularDistancia(arbol.getDatoRaiz().texto, elementoABuscar);


    //Si la distancia está dentro del umbral, agrego el resultado
    if (distancia <= umbral)
    {
        var nodo = arbol.getDatoRaiz();

        //agrego una copia de DatoDistancia donde la "distancia" es la distancia respectoa la busqueda
        //esto evita mostrar la distancia almacenada en el nodo, que es la distancai al padre
        collected.Add(new DatoDistancia (distancia, nodo.texto,nodo.descripcion));
    }


    // Recorro recursivamente todos los hijos del nodo actual
    foreach (var hijo in arbol.getHijos())
    {
        Buscar(hijo, elementoABuscar, umbral, collected); // Llamada recursiva para cada hijo
    }
}
```

El método **Buscar** recorre el árbol de forma recursiva para encontrar los nodos cuyo texto se parezca al término indicado (elementoABuscar), dentro de un margen definido por umbral. Primero valida que el árbol y la lista de resultados no sean nulos, y reemplaza el término por una cadena vacía si eso ocurre. Luego calcula la distancia entre el texto del nodo y el término de búsqueda usando **CalcularDistancia**. Si la distancia es menor o igual al umbral, agrega un nuevo **DatoDistancia** a la lista de resultados. Finalmente, repite el proceso con cada hijo del nodo para recorrer todo el árbol.

```
// Método que agrega un dato al árbol respetando la distancia calculada
public void AgregarDato(ArbolGeneral<DatoDistancia> arbol, DatoDistancia dato)
    {
        // Si el árbol es nulo no hago nada
        if (arbol == null)
        {
            return;
        }

        // Calculo la distancia entre la raiz actual y el nuevo dato a insertar
        int distancia = CalcularDistancia(arbol.getDatoRaiz().texto, dato.texto);

        // Recorro los hijos para ver si ya existe una rama con esa misma distancia
        foreach (var hijo in arbol.getHijos())
        {
            // Si encuentro un hijo cuya etiqueta de distancia coincide
            // inserto recursivamente en ese hijo (descenso por la rama)
            if (hijo.getDatoRaiz().distancia == distancia)
            {
                AgregarDato(hijo, dato);
                return; //Termino porque ya inserté
            }
        }

        // Si no encontré un hijo con esa distancia, creo un nuevo nodo y lo agrego como hijo
        var datoNuevo = new DatoDistancia(distancia, dato.texto, dato.descripcion);// Creo el DatoDistancia con la distancia al padre
        var nuevoNodo = new ArbolGeneral<DatoDistancia>(datoNuevo);// Creo el nodo del árbol
        arbol.agregarHijo(nuevoNodo);// Lo agrego como hijo de la raíz actual
    }
```

El método **AgregarDato** inserta un nuevo elemento en el árbol BK, respetando su estructura métrica. Primero valida que el árbol no sea nulo para evitar errores. Luego calcula la distancia entre el texto del nodo actual y el del dato a insertar, usando el método **CalcularDistancia**. Después recorre los hijos del nodo actual buscando si ya existe una rama con esa misma distancia. Si la encuentra, llama recursivamente a **AgregarDato** sobre ese subárbol y finaliza. En caso de no encontrar un hijo con la distancia coincidente, se procede a crear una nueva rama. Se instancia un **DatoDistancia** con el valor de la distancia, se utiliza para crear un nuevo nodo **ArbolGeneral** y este se establece como un nuevo hijo del nodo actual.

```
public String Consulta1(ArbolGeneral<DatoDistancia> arbol)
{
    //llamo al metodo recursivo para
    string resultado = recolectarHojas(arbol);

    //devuelve la cadena de texto completa que se generó
    return resultado;


}

private string recolectarHojas(ArbolGeneral<DatoDistancia> arbol)
{

    //se inicializa un string vacío para ir acumulando las hojas encontradas en esta rama
    string hojasEncontradas = "";

    //si el nodo actual es una hoja...
    if (arbol.esHoja())
    {
        //se añade la información del nodo al string y se agrega un salto de linea
        hojasEncontradas += arbol.getDatoRaiz().ToString() + "\n";
    }

    //si el nodo actual no es una hoja...
    else
    {
        //se recorre cada uno de sus hijos
        foreach (ArbolGeneral<DatoDistancia> hijo in arbol.getHijos())
        {
            //elresultado de la llamada(las hojas encontradas en el subárbol del hijo) se concatena al string "hojasEncontradas"
            hojasEncontradas += recolectarHojas(hijo);
        }

    }

    //devuelve el string con las hojas encontradas en este nivek del arbol
    return hojasEncontradas;
}
```

El método **Consulta1** recorre el árbol y devuelve una lista con todos los nodos hoja. No realiza el recorrido directamente, sino que delega esa tarea a un método auxiliar recursivo llamado **recolectarHojas**. El método **recolectarHojas** utiliza un recorrido en profundidad (DFS). Si el nodo actual no tiene hijos, agrega su texto al resultado. Si tiene hijos, se llama recursivamente sobre cada uno, concatenando los resultados para formar la salida final. Al terminar el recorrido, **recolectarHojas** devuelve una cadena con todos los nodos hoja encontrados, que luego **Consulta1** retorna como resultado.

```
public String Consulta2(ArbolGeneral<DatoDistancia> arbol)
{
    // Lista para guardar las líneas de texto de cada camino encontrado.
    List<string> todosLosCaminosEncontrados = new List<string>();

    // Lista para construir el camino actual durante la recursión.
    // Usa 'DatoDistancia' para tener acceso a la distancia de cada nodo.
    List<DatoDistancia> caminoActual = new List<DatoDistancia>();

    // Inicia el proceso recursivo.
    RecolectarCaminos(arbol, caminoActual, todosLosCaminosEncontrados);

    // Une todas las líneas de texto en un solo string final.
    string resultadoFinal = "";
    foreach (string linea in todosLosCaminosEncontrados)
    {
        resultadoFinal += linea + "\n"; //se añade cadacamino
    }
    return resultadoFinal;
}
```

```
private void RecolectarCaminos(ArbolGeneral<DatoDistancia> arbol, List<DatoDistancia> caminoActual, List<string> todosLosCaminosEncontrados)
{
    // 1. Agrega el objeto completo (con texto y distancia) al camino.
    caminoActual.Add(arbol.getDatoRaiz());

    if (arbol.esHoja())
    {
        // 2. Si es hoja, construye el string final para este camino.
        string caminoCompleto = "";
        for (int i = 0; i < caminoActual.Count; i++)
        {
            DatoDistancia dato = caminoActual[i];

            // Formatea cada nodo del camino con su distancia guardada.
            // La distancia de la raíz debe ser 0 para que aparezca como (0).
            caminoCompleto += "(" + dato.distancia + ") " + dato.texto;

            // se agrega una coma pero solo si no es el último elemento.
            if (i < caminoActual.Count - 1)
            {
                caminoCompleto += ", ";
            }
        }
        //se guarda el camino formateado en la lista de resultados finales
        todosLosCaminosEncontrados.Add(caminoCompleto);
    }
    else
    {
        //Si no es hoja, la recursión continúa en cada hijo.
        foreach (ArbolGeneral<DatoDistancia> hijo in arbol.getHijos())
        {
            RecolectarCaminos(hijo, caminoActual, todosLosCaminosEncontrados);
        }
    }

    //despues de haber visitado un nodo y todos sus descendientes (es decir, al salir de la llamada recursiva),
    //se elimina el nodo actual del camino
    //esto es "como dar un paso atras" para poder explorar otras ramas del arbol
    //si no hicieramos esto, los camunos de diferentes ramas se mezclarian.
    caminoActual.RemoveAt(caminoActual.Count - 1);
}
```

El método **Consulta2** genera un reporte con todos los caminos que van desde la raíz hasta cada hoja del árbol. Para hacerlo, utiliza el método recursivo

RecolectarCaminos, que aplica una estrategia de backtracking para recorrer la estructura. Primero, Consulta2 prepara dos listas: una para guardar los caminos finales y otra temporal (**caminoActual**) que se va completando a medida que el recorrido avanza. Luego, llama a **RecolectarCaminos** para iniciar el proceso. En **RecolectarCaminos**, cada nodo visitado se agrega al camino actual. Si el nodo es una hoja, se construye una cadena con el recorrido completo y se guarda en la lista de resultados. Si no lo es, el método continúa recursivamente con sus hijos. Al volver de una rama, se elimina el último nodo agregado, permitiendo explorar otras rutas correctamente. Al finalizar, Consulta2 une todos los caminos recolectados en un único texto, separándolos con saltos de línea para generar el reporte final.

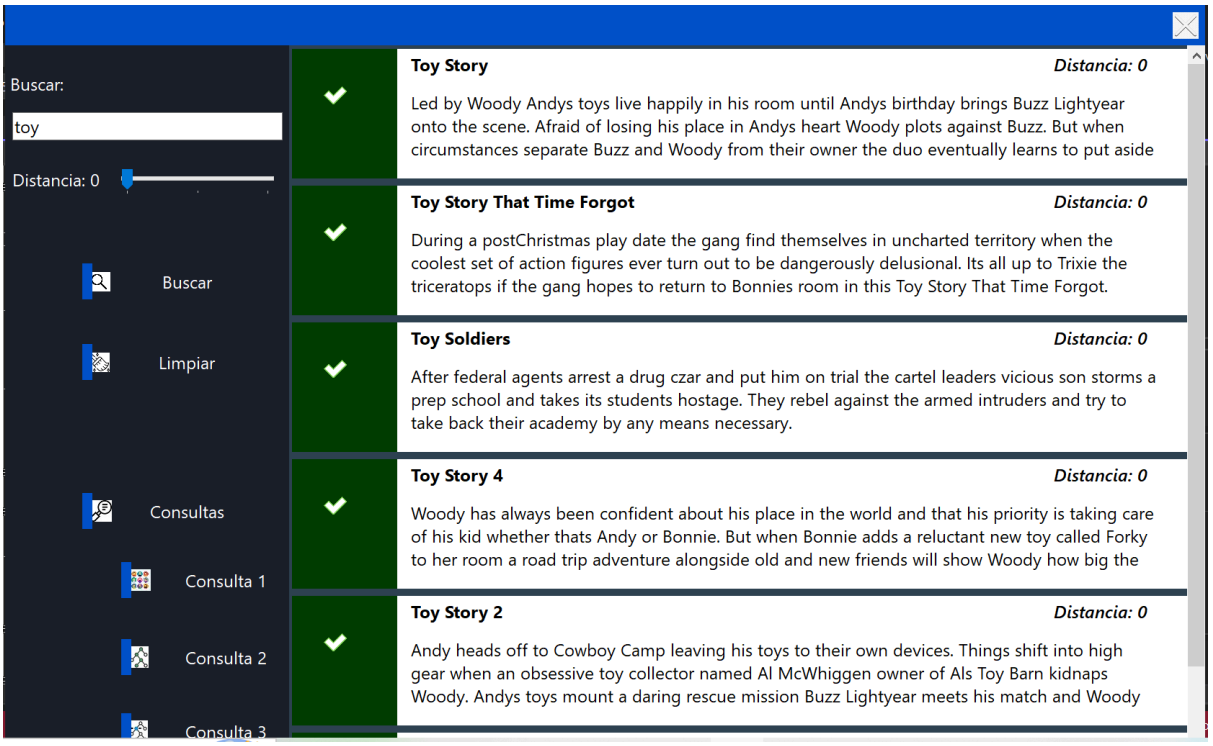
```
public String Consulta3(ArbolGeneral<DatoDistancia> arbol)
{
    //si el arbol no existe o esta vacio, devuelve un mensaje y termina
    if (arbol == null || arbol.esVacio())
    {
        return "El árbol está vacío.";
    }

    string resultadoFinal = "";

    //se crea una cola que almacenará los nodos a visitar
    Cola<ArbolGeneral<DatoDistancia>> cola = new Cola<ArbolGeneral<DatoDistancia>>();

    //se encola el primer nodo (la raiz del arbol) parainiciar el proceso
    cola.encolar(arbol);

    //se inicializa un contador para llevar la cuenta del nivel actual
    int nivel = 0;


  //mientras la cola no este vacía...
  while (!cola.esVacia())
  {
      // Se obtiene el numero de nodos que hay actualmente en la cola
      int nodosEnNivel = cola.cantidadElementos();
      resultadoFinal += "Nodos en el Nivel " + nivel + "\n";

      //lista temporal para guardar los textos de los nodos
      List<string> textosDelNivel = new List<string>();


      //este bucle se ejecutará "nodosEnNivel" veces, procesando cada nodo del nivel actual
      for (int i = 0; i < nodosEnNivel; i++)
      {
          //se saca el siguiente nodo de la cola para procesarlo
          ArbolGeneral<DatoDistancia> nodoActual = cola.desencolar();
          DatoDistancia dato = nodoActual.getDatoRaiz();

          //se formatea el teto del nodo yse guarda en la lista temporal
          string textoFormateado = "(" + dato.distancia + ") " + dato.texto;
          textosDelNivel.Add(textoFormateado);

          //se recorren todos los hijos del nodo actual
          foreach (ArbolGeneral<DatoDistancia> hijo in nodoActual.getHijos())
          {
              // Se encolan
              cola.encolar(hijo);
          }
      }
```

```
        //se recorre la lista de textos del nivel que acabamos de procesar
        for (int i = 0; i < textosDelNivel.Count; i++)
        {
            //se agrega cada texto al resultado final
            resultadoFinal += textosDelNivel[i];

            //si no es el ultimo elemento se añade una coma y un espacio
            if (i < textosDelNivel.Count - 1)
            {
                resultadoFinal += ", ";
            }
        }
        resultadoFinal += "\n";

        //se incrementa el contador de nivel para la siguiente iteración
        nivel++;
    }

    //una vez que la cola esta vacia, se han visitado todos los nodos
    //se devuelve el string con el resultado completo

    return resultadoFinal;
}
```

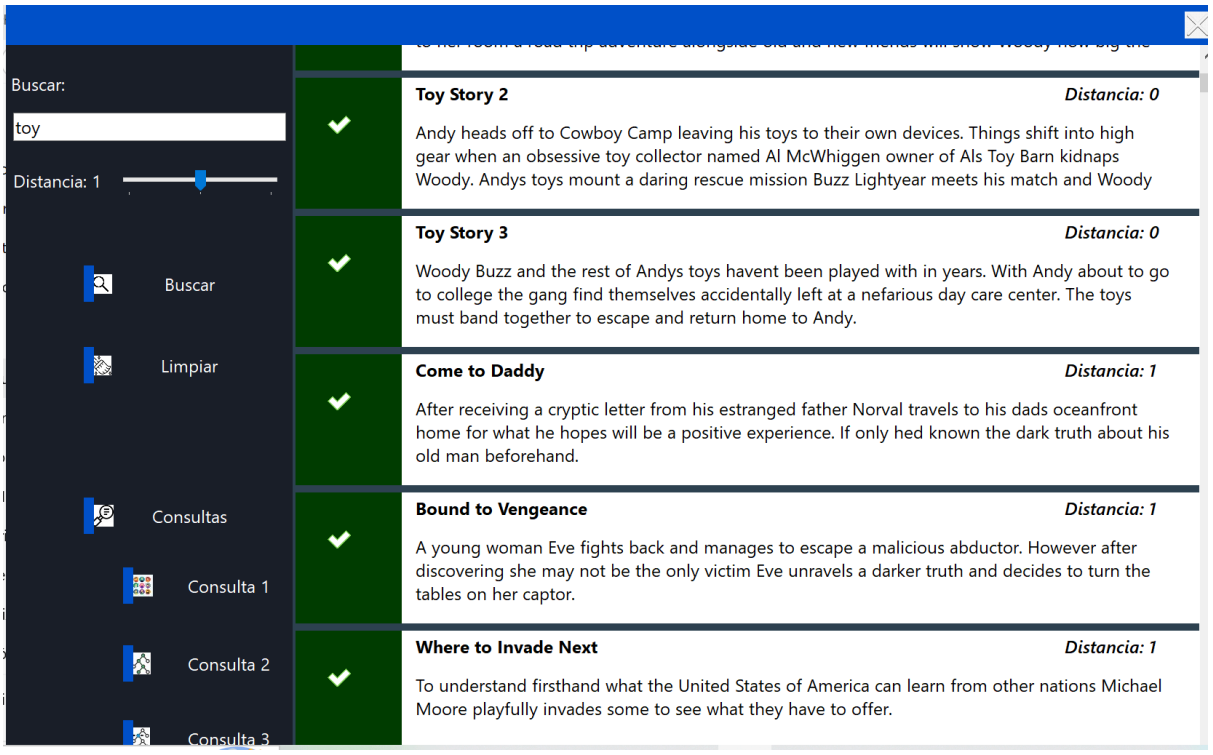
El método **Consulta3** genera un reporte del árbol y muestra los nodos organizados por nivel de profundidad. Para hacerlo, aplica un recorrido por niveles o **búsqueda en anchura (BFS)**, utilizando una cola como estructura auxiliar. Primero, verifica que el árbol no esté vacío; si lo está, devuelve un mensaje. Luego, inicializa una cola y agrega el nodo raíz como punto de partida. El bucle while se ejecuta mientras haya nodos en la cola. En cada iteración, procesa un nivel completo del árbol. Para ello, determina cuántos nodos hay actualmente en la cola (nodosEnNivel) y usa un for para recorrer solo esos elementos. Dentro del for, se desencola un nodo, se formatea su información en texto y se guarda en una lista temporal (textosDelNivel). Después, se encolan todos sus hijos, que serán procesados en el siguiente nivel. Cuando se terminan de procesar los nodos del nivel actual, se construye una línea de texto con todos los elementos de textosDelNivel, separados por comas. Luego se incrementa el contador de nivel y el proceso continúa hasta que la cola queda vacía y el recorrido ha cubierto todo el árbol.

---

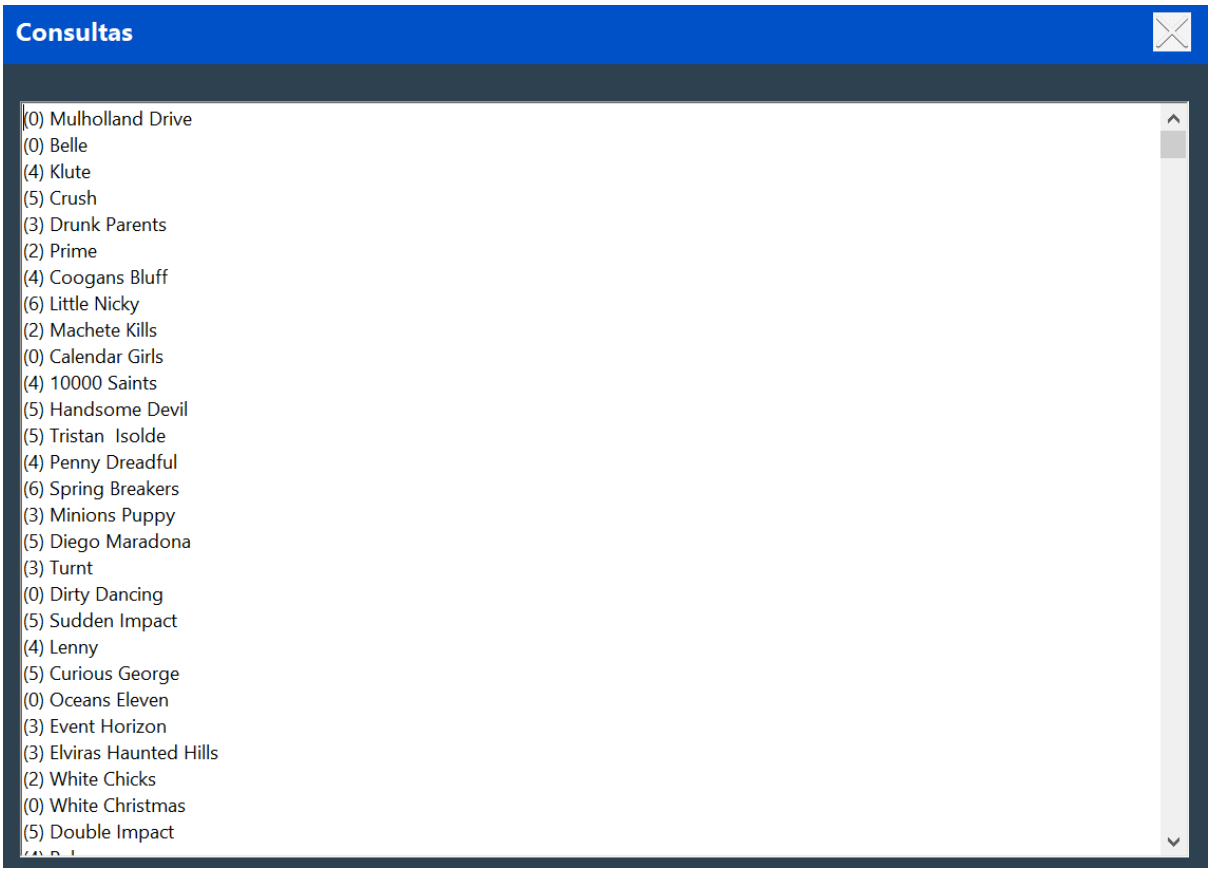
ii. interfaz de usuario final



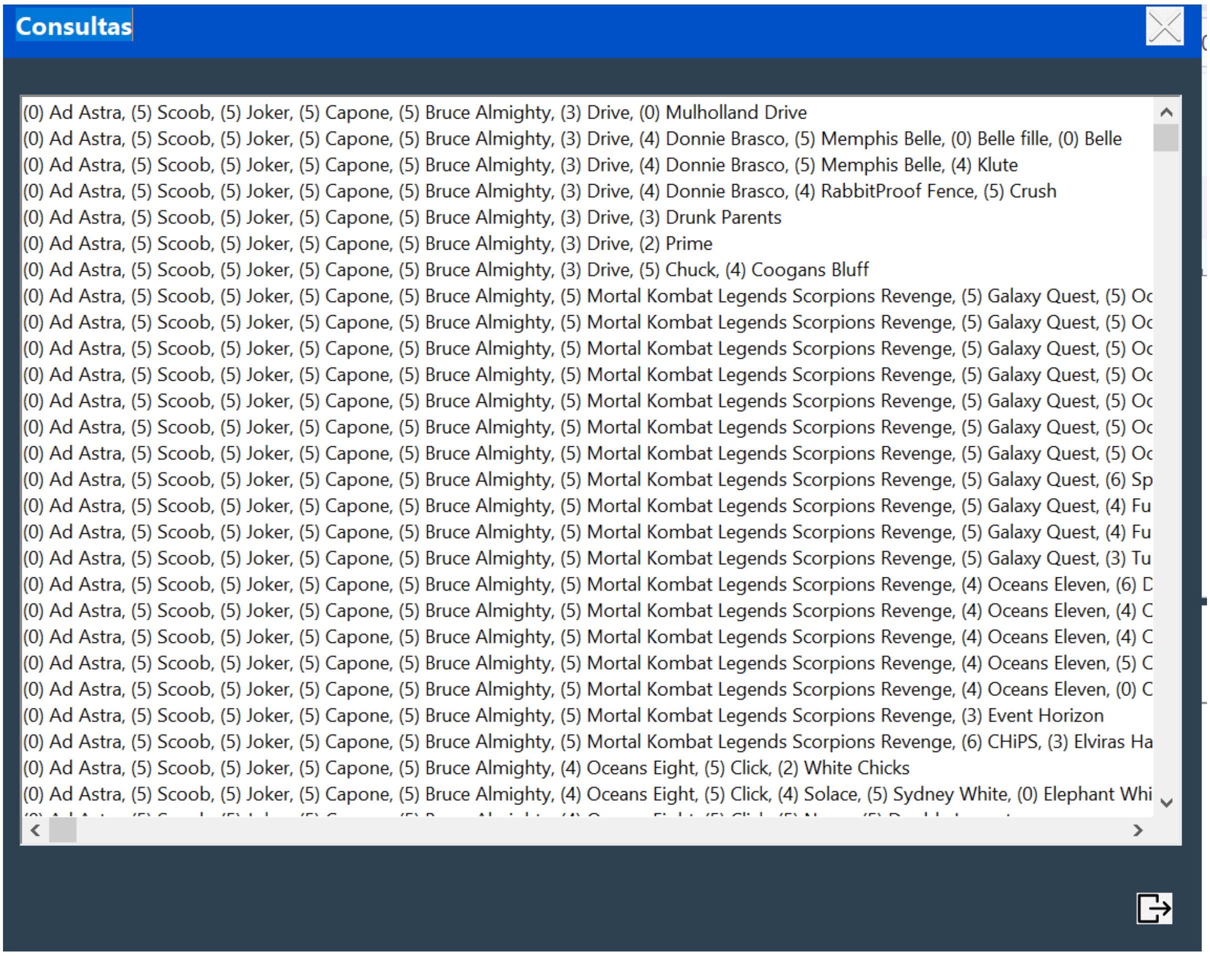
Distancia de Levenshtein 0



Distancia de Levenshtein 1



Consulta 1


Consultas
(0) Ad Astra, (5) Scoob, (5) Joker, (5) Capone, (5) Bruce Almighty, (3) Drive, (0) Mulholland Drive
(0) Ad Astra, (5) Scoob, (5) Joker, (5) Capone, (5) Bruce Almighty, (3) Drive, (4) Donnie Brasco, (5) Memphis Belle, (0) Belle fille, (0) Belle
(0) Ad Astra, (5) Scoob, (5) Joker, (5) Capone, (5) Bruce Almighty, (3) Drive, (4) Donnie Brasco, (5) Memphis Belle, (4) Klute
(0) Ad Astra, (5) Scoob, (5) Joker, (5) Capone, (5) Bruce Almighty, (3) Drive, (4) Donnie Brasco, (4) RabbitProof Fence, (5) Crush
(0) Ad Astra, (5) Scoob, (5) Joker, (5) Capone, (5) Bruce Almighty, (3) Drive, (3) Drunk Parents
(0) Ad Astra, (5) Scoob, (5) Joker, (5) Capone, (5) Bruce Almighty, (3) Drive, (2) Prime
(0) Ad Astra, (5) Scoob, (5) Joker, (5) Capone, (5) Bruce Almighty, (3) Drive, (5) Chuck, (4) Coogans Bluff
(0) Ad Astra, (5) Scoob, (5) Joker, (5) Capone, (5) Bruce Almighty, (5) Mortal Kombat Legends Scorpions Revenge, (5) Galaxy Quest, (5) Oc
(0) Ad Astra, (5) Scoob, (5) Joker, (5) Capone, (5) Bruce Almighty, (5) Mortal Kombat Legends Scorpions Revenge, (5) Galaxy Quest, (5) Oc
(0) Ad Astra, (5) Scoob, (5) Joker, (5) Capone, (5) Bruce Almighty, (5) Mortal Kombat Legends Scorpions Revenge, (5) Galaxy Quest, (5) Oc
(0) Ad Astra, (5) Scoob, (5) Joker, (5) Capone, (5) Bruce Almighty, (5) Mortal Kombat Legends Scorpions Revenge, (5) Galaxy Quest, (5) Oc
(0) Ad Astra, (5) Scoob, (5) Joker, (5) Capone, (5) Bruce Almighty, (5) Mortal Kombat Legends Scorpions Revenge, (5) Galaxy Quest, (5) Oc
(0) Ad Astra, (5) Scoob, (5) Joker, (5) Capone, (5) Bruce Almighty, (5) Mortal Kombat Legends Scorpions Revenge, (5) Galaxy Quest, (5) Oc
(0) Ad Astra, (5) Scoob, (5) Joker, (5) Capone, (5) Bruce Almighty, (5) Mortal Kombat Legends Scorpions Revenge, (5) Galaxy Quest, (5) Oc
(0) Ad Astra, (5) Scoob, (5) Joker, (5) Capone, (5) Bruce Almighty, (5) Mortal Kombat Legends Scorpions Revenge, (5) Galaxy Quest, (6) Sp
(0) Ad Astra, (5) Scoob, (5) Joker, (5) Capone, (5) Bruce Almighty, (5) Mortal Kombat Legends Scorpions Revenge, (5) Galaxy Quest, (4) Fu
(0) Ad Astra, (5) Scoob, (5) Joker, (5) Capone, (5) Bruce Almighty, (5) Mortal Kombat Legends Scorpions Revenge, (5) Galaxy Quest, (4) Fu
(0) Ad Astra, (5) Scoob, (5) Joker, (5) Capone, (5) Bruce Almighty, (5) Mortal Kombat Legends Scorpions Revenge, (5) Galaxy Quest, (3) Tu
(0) Ad Astra, (5) Scoob, (5) Joker, (5) Capone, (5) Bruce Almighty, (5) Mortal Kombat Legends Scorpions Revenge, (4) Oceans Eleven, (6) D
(0) Ad Astra, (5) Scoob, (5) Joker, (5) Capone, (5) Bruce Almighty, (5) Mortal Kombat Legends Scorpions Revenge, (4) Oceans Eleven, (4) C
(0) Ad Astra, (5) Scoob, (5) Joker, (5) Capone, (5) Bruce Almighty, (5) Mortal Kombat Legends Scorpions Revenge, (4) Oceans Eleven, (4) C
(0) Ad Astra, (5) Scoob, (5) Joker, (5) Capone, (5) Bruce Almighty, (5) Mortal Kombat Legends Scorpions Revenge, (4) Oceans Eleven, (5) C
(0) Ad Astra, (5) Scoob, (5) Joker, (5) Capone, (5) Bruce Almighty, (5) Mortal Kombat Legends Scorpions Revenge, (4) Oceans Eleven, (0) C
(0) Ad Astra, (5) Scoob, (5) Joker, (5) Capone, (5) Bruce Almighty, (5) Mortal Kombat Legends Scorpions Revenge, (3) Event Horizon
(0) Ad Astra, (5) Scoob, (5) Joker, (5) Capone, (5) Bruce Almighty, (5) Mortal Kombat Legends Scorpions Revenge, (6) CHiPS, (3) Elviras Ha
(0) Ad Astra, (5) Scoob, (5) Joker, (5) Capone, (5) Bruce Almighty, (4) Oceans Eight, (5) Click, (2) White Chicks
(0) Ad Astra, (5) Scoob, (5) Joker, (5) Capone, (5) Bruce Almighty, (4) Oceans Eight, (5) Click, (4) Solace, (5) Sydney White, (0) Elephant Whi

Consulta 2

Consulta 3

Errores detectados en la ejecución del programa

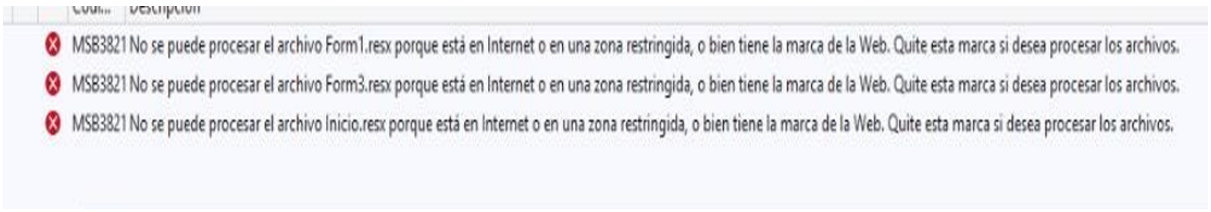
I. Error en .NET

Al intentar ejecutar el proyecto en SharpDevelop, se presentó un mensaje de error indicando que el proyecto era incompatible. Tras investigar, se determinó que el proyecto fue desarrollado utilizando **.NET 6.0**, lo que requiere esta versión específica del framework para su correcta ejecución. Dado que SharpDevelop no soporta .NET 6.0 de manera nativa, no fue posible ejecutar el proyecto en dicho entorno. Por esta razón, se optó por utilizar **Visual Studio en su versión más reciente**, la cual es totalmente compatible con .NET 6.0, permitiendo así que la aplicación pueda realizar correctamente la indexación de los archivos CSV.

II.Errores en archivos .resx

También pueden aparecer errores del tipo:

• "No se puede procesar el archivo [nombre].resx porque está en Internet o en una zona restringida, o bien tiene la marca de la Web".



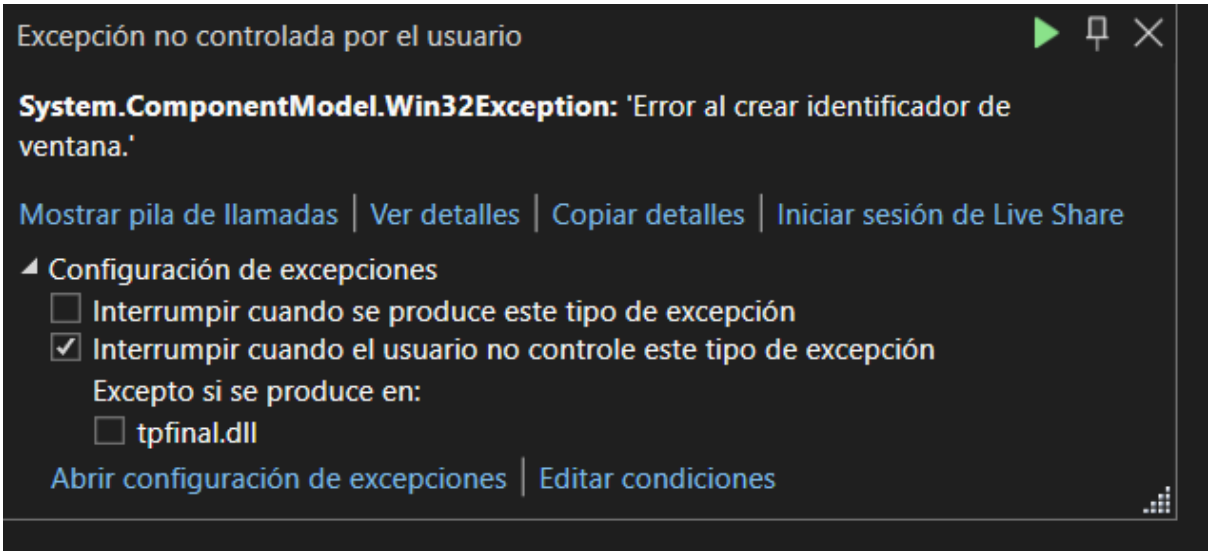
Este error se debe a que los archivos .resx (archivos de recursos del formulario) han sido descargados desde la web y Windows los marca como inseguros por defecto.
Solución:
• Hacer clic derecho sobre el archivo afectado (.resx) → Propiedades.
• En la pestaña General, en la parte inferior, se encuentra la opción “Desbloquear”. Debe tildarse esta casilla para eliminar la marca de seguridad aplicada por el sistema.
• Aceptar los cambios y volver a compilar el proyecto.

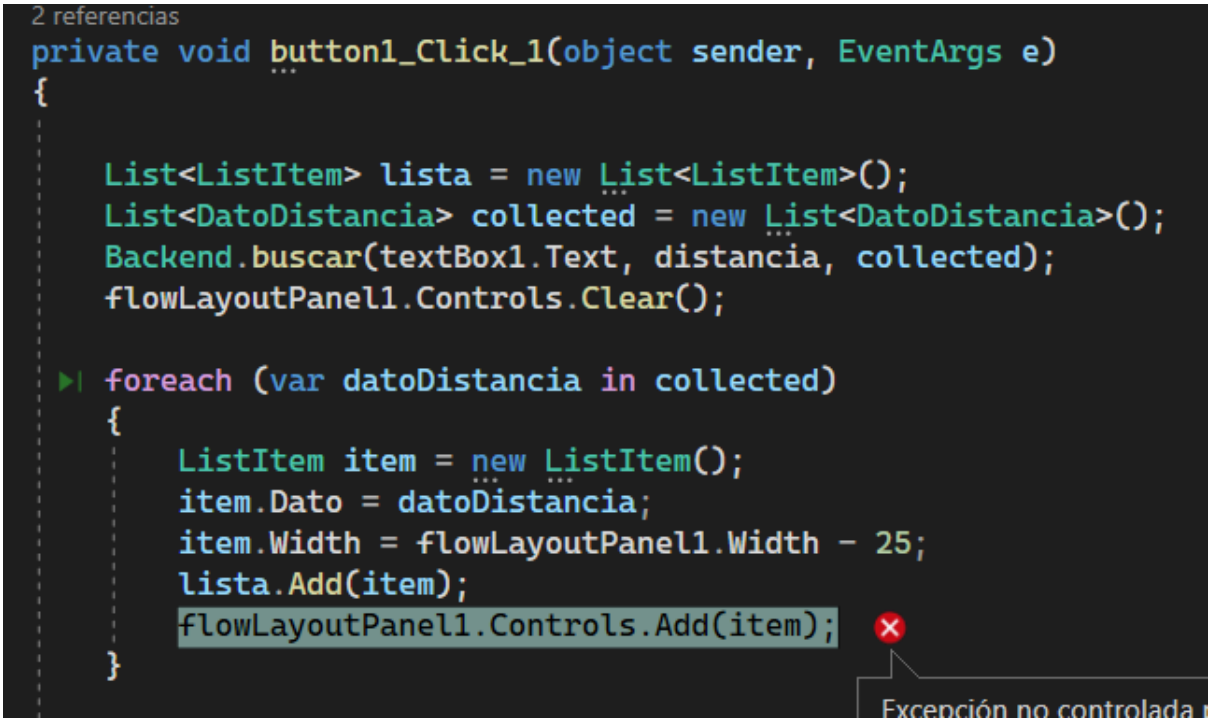
III.Otros problemas encontrados

El problema ocurre en el momento de mostrar los resultados de la búsqueda para distancia 2. Cada vez que busco un texto dentro de la base movie.csv (que contiene 10.000 películas), el programa recorre todo el árbol y agrega un control gráfico (**ListItem**) por cada coincidencia dentro del **FlowLayoutPanel**.

De esta forma, al buscar cualquier palabra, se empiezan a crear miles de controles en memoria. Esto hace que el consumo de CPU y RAM aumente rápidamente, ya que Windows **Forms** debe manejar todos esos objetos gráficos al mismo tiempo.

El resultado final es que, al superar el límite de recursos que Windows permite para crear identificadores de ventana, el programa termina lanzando la excepción en la clase **Form1**: **System.ComponentModel.Win32Exception: Error al crear identificador de ventana.**





Se debe modificar el código en el botón "Buscar" (button1_Click_1) para que el programa a lo sumo no lance la excepción:

• Una vez que la búsqueda terminó, se debe usar el método. Take() de LINQ para tomar solo una porción manejable de esa lista (por ejemplo, los primeros 100) y solo entonces crea los controles ListItem para mostrarlos en el FlowLayoutPanel.

Obs: Al hacer esto también muchos resultados de la búsqueda no serán mostrados.

Problemas al realizar el código:

En la clase Estrategia, la parte más complicada fue el cálculo de la distancia. Al buscar por ejemplo palabra, “toy”, el programa devolvía la película “Toy Story” pero con distancia 1 en lugar de 0. Esto ocurría porque siempre aplicaba Levenshtein sin comprobar coincidencias exactas. Lo solucioné agregando una condición previa que devuelve 0 cuando las palabras son iguales:

if (s1 == s2) return 0;

Otro problema era que si la palabra tenía espacios al inicio o al final, estos se tomaban como parte de la distancia. Para corregirlo, incorporé un Trim() antes del ToLower(). Noté además que, si no se ingresaba ninguna palabra, el programa devolvía películas de todas formas. Esto lo resolví agregando una condición que valida si alguna de las cadenas está vacía o nula, en cuyo caso retorna un valor fuera de rango (1000) para que no se muestre ningún resultado:

if (string.IsNullOrWhiteSpace(str1) || string.IsNullOrWhiteSpace(str2))
return 1000;

Para **Consulta1**, el problema surgió en la recursividad. Al principio, intenté usar una variable de instancia de la clase para ir guardando las hojas encontradas. Sin embargo, como llamaba al método más de una vez, los nuevos resultados se añadían a los anteriores, duplicando la información. La solución fue modificarlo para que, en lugar de guardar los resultados en una variable externa, devolviera directamente lo que encontraba. El método ahora devuelve un string y cada llamada recursiva concatena los resultados de sus hijos. Así, se eliminó la dependencia de un estado externo.

// Se acumula el resultado devuelto por las llamadas recursivas
hojasEncotradas += recolectarHojas(hijo);

En **Consulta2**, al explorar una rama y luego pasar a otra, los nodos de la primera se quedaban en la lista del "camino actual", corrompiendo las rutas siguientes. La solución fue implementar la técnica de backtracking. Después de que una llamada recursiva haya explorado por completo un nodo y todos sus descendientes, se elimina ese mismo nodo de la lista del camino. Este paso de "retroceso" limpia la ruta y permite que la exploración de las ramas hermanas comience desde el estado correcto.

caminoActual.RemoveAt(caminoActual.Count - 1);

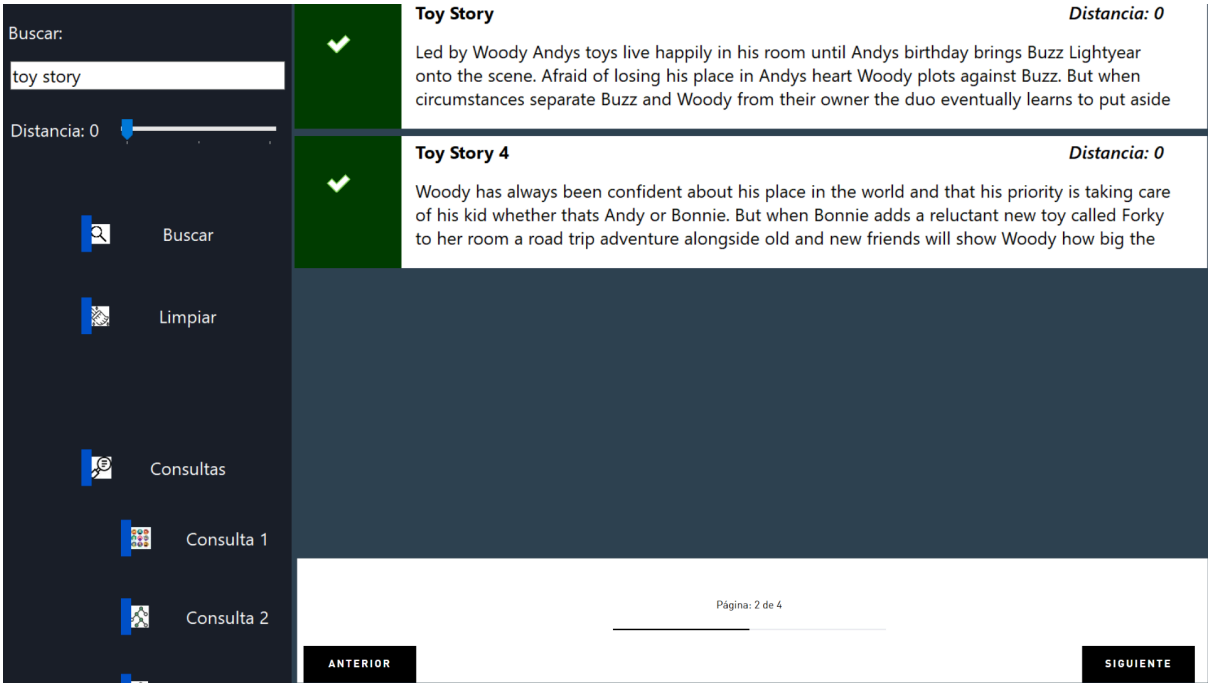
En **Consulta3** el bucle while sobre la cola procesaba todos los nodos de forma continua, sin agruparlos y no diferenciaba dónde terminaba un nivel y empezaba el siguiente. La solución fue utilizar un bucle anidado. Antes de empezar a procesar los nodos, se guarda el tamaño actual de la cola en una variable. El bucle interior se

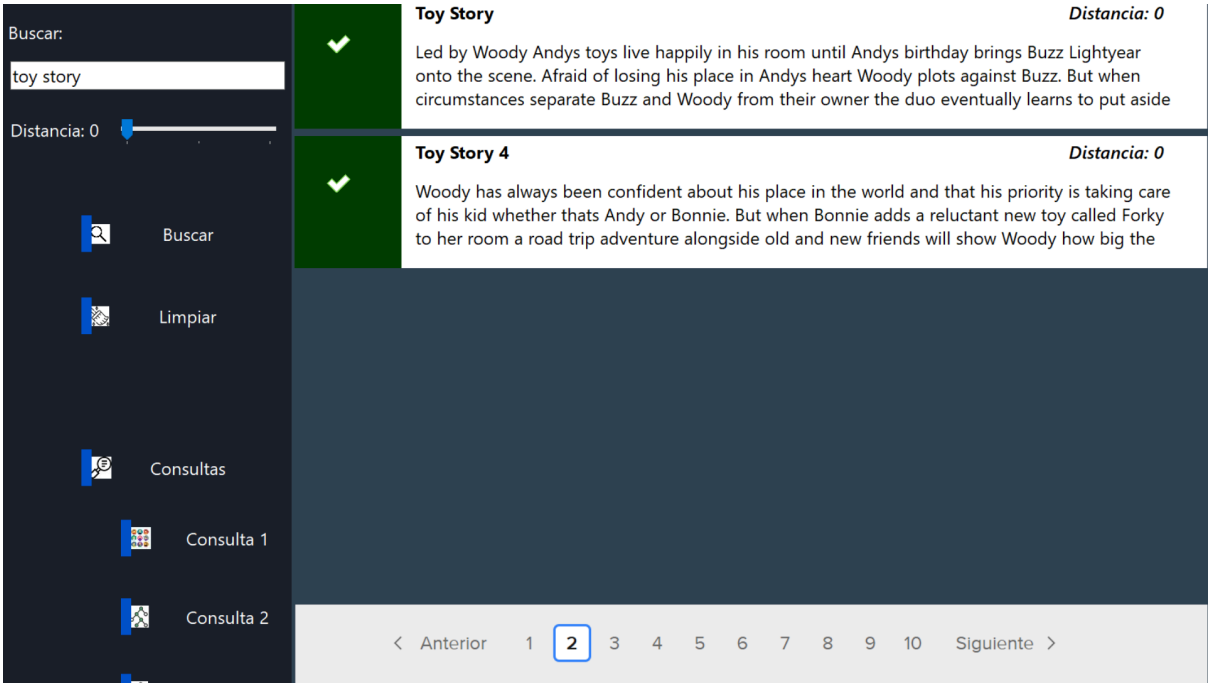
ejecuta exactamente esa cantidad de veces. Esto me permitió procesar todos los nodos de un nivel, y solo esos, antes de pasar al siguiente, logrando así separación.

```
int nodosEnNivel = cola.cantidadElementos();
        for (int i = 0; i < nodosEnNivel; i++)
        {
            // ... procesar el nodo y encolar a sus hijos  }
```

Posibles Mejoras

En la versión actual, la interfaz muestra solo los primeros 100 resultados para evitar la sobrecarga de controles gráficos. Una mejora posible sería incorporar un sistema con botones “Siguiente” y “Anterior”, permitiendo recorrer de a tramamos todos los resultados sin afectar el rendimiento de la aplicación.





Conclusión

La implementación del árbol BK mediante la clase *Estrategia* constituyó una solución práctica y eficiente para la búsqueda de coincidencias aproximadas. Los métodos “Agregar”, “Buscar”, “consulta1”, “consulta2” y “consulta3” se desarrollaron aplicando de manera directa los conceptos de estructuras de datos y recursividad estudiados en la materia.

En síntesis, la experiencia permitió integrar la teoría con la práctica, verificando el correcto funcionamiento del sistema y consolidando la comprensión de los algoritmos y su aplicación en contextos reales.