

Buscador_Zanahorias

Link del repositorio: https://github.com/JulianSalinas/Buscador_Zanahorias

Proyecto Corto II: Inteligencia Artificial

El propósito es aplicar dos tipos de algoritmos de búsqueda a un determinado problema. Para simplificar la explicación del mismo, se abstrae mediante el uso de un conejo que debe encontrar una cantidad determinada de zanahorias dentro del huerto.

Terminología

1. Huerto: Matriz de `n x m` Parcela: Posición `(i, j)` de la matriz.
2. Zanahoria: Posición `(i, j)` de la matriz que contiene la letra `Z`.
3. Flecha: Posición `(i, j)` de la matriz que contiene `>`, `<`, `A`, o `V`.

El primer algoritmo es `A*`. En este el conejo debe encontrar las zanahorias moviéndose con base en una heurística aplicada a un rango de visión, es decir, con una cantidad delimitada de parcelas. El objetivo es que la heurística permita al conejo encontrar las zanahorias que requiere en la menor cantidad de pasos.

El segundo se trata de un algoritmo `genético`. En este se limita al conejo a moverse en una sola dirección. Para que pueda girar debe encontrar una flecha en la parcela que le indique hacia donde. El propósito es que el algoritmo coloque flechas en el huerto para que el conejo pueda moverse y encontrar todas las zanahorias. En caso de que se generen varias soluciones para encontrarlas todas, se seleccionan las que resuelven el problema en la menor cantidad de pasos.

Instalación

Dependencias

Es necesario contar con `Python 3` para poder realizar la instalación. Además, se requieren algunas dependencias que se pueden instalar utilizando `pip`:

1. Se abre un terminal con permisos de administrador
2. Se ejecutan los siguientes comandos

```
sudo pip3 install numpy
```

```
sudo pip3 install pandas
```

Instalación desde el código fuente

1. Se debe abrir una terminal con permisos de administrador
2. Se debe navegar hasta la ruta del código fuente, al nivel que se encuentre el archivo `setup.py`
3. Se ejecuta el siguientes comandos:

```
sudo python3 setup.py install
```

Instalación utilizando pip

1. Se debe abrir una terminal con permisos de administrador
2. Se ejecuta el siguiente comando:

```
sudo pip3 install tec.ic.ia.pc2
```

Manual de uso

Ejecución desde consola

Una vez instalado se pueden utilizar el algoritmo `A*` de la siguiente forma:

```
python3 main.py --a-estrella --vision n --zanahoria n --tablero-inicial entrada.txt
```

```
ejemplo$ python 3 main.py main.py --a-estrella --vision 4 --zanahoria 2 --tablero-inicial entrada.txt
```

El algoritmo `genético` se utiliza mediante el comando

```
python main.py --genetico --individuos n --generaciones n --zanahoria n --politica 1 --mutaciones n --derecha --tablero-inicial entrada.txt
```

```
ejemplo$ python main.py --genetico --individuos 20 --generaciones 50 --zanahoria 2 --politica 1 --mutaciones 10 --derecha --tablero-inicial /home/usuario/Desktop/entrada.txt
```

Parámetros

1. **--a-estrella:** Indica que se desea ejecutar el algoritmo `A*`.
2. **--genetico:** Indica que se desea ejecutar el algoritmo `genético`.
3. **--vision:** Indica la cantidad de parcelas que el conejo puede observar desde su posición. Este solo aplica para el algoritmo `A*`.
4. **--zanahoria:** Indica la cantidad de zanahorias que el conejo debe encontrar para terminar la ejecución del algoritmo `A*`.
5. **--derecha, --izquierda, --arriba o --abajo:** Indica hacia que lado tiene que avanzar el conejo inicialmente. Se debe indicar solo para el algoritmo `genético`.
6. **--individuos:** Indica de genes o individuos que el algoritmo `genético` debe mantener en cada generación.
7. **--generaciones:** Indica cuántas generaciones como máximo debe realizar el algoritmo `genético`.

8. **--politica:** Indica que politica de cruce se debe utilizar en el algoritmo `genético`. Se especifica `1` para el cruce con `corte en un punto` y `2` para cruces con `corte en dos puntos`.
9. **--mutaciones:** Indica por medio de un `número de 0 a 100`, cuál es la probabilidad de mutación para un individuo.
10. **--semilla:** Si se desea obtener los mismos resultados al usar los mismos parámetros, se debe usar una misma semilla para las distintas corridas.
11. **--tablero- inicial:**

Nombre del archivo de texto que contiene el tablero inicial.

1. `C` en mayúscula, identifica la posición del conejo. Sólo podrá haber uno por tablero.
2. `Z` en mayúscula, identifica la posición de una zanahoria. Puede haber múltiples zanahorias.
3. o espacio. Indica una posición en el tablero por la que el conejo puede transitar.
4. `<` es el símbolo que indica un cambio de dirección hacia la izquierda.
5. `>` es el símbolo que indica un cambio de dirección hacia la derecha.
6. `A` en mayúscula, indica un cambio de dirección hacia arriba.
7. `V` : en mayúscula, indica un cambio de dirección hacia abajo.
8. Cambio de línea: No tiene ninguna interpretación en el programa más que separar las filas

Importar en un archivo

Para utilizar el módulo instalado se puede importar de la siguiente forma desde cualquier archivo `.py` o desde el shell de `Python`:

```
from tec.ic.ia.pc2.g03 import carrot_finder
```

Algoritmo de A*

El objetivo principal fue desarrollar un algoritmo de búsqueda que se basa en la técnica de A*, en un ambiente donde se cuenta con un tablero de juego, un conejo como punto de partida y una serie de zanahorias que cumplan la función de metas u objetivos dentro del algoritmo a desarrollar.

Flujo de implementación

En esta sección, se pretende mostrar una pequeña lista de pasos que se llevaron a cabo durante el desarrollo del proyecto, con la finalidad de tener una idea más clara del comportamiento del algoritmo implementado.

1. En primera instancia, se debe tener claro que el algoritmo necesita tanto de un tablero de juego, donde se encuentran ubicadas las zanahorias y el conejo. También un parámetro que indique cuántas zanahorias son necesarias para que el conejo se de por satisfecho y además, un número indicando el rango de visión que tiene el conejo a su alrededor.
2. Una vez que dichos parámetros son ingresados, el algoritmo entra en un ciclo (while) el cual mantiene en ejecución la función principal hasta que el conejo logre comerse todas las zanahorias que son necesarias. En contraparte, si el conejo no lograra encontrarlas, el sistema deberá abortarse por medio de la combinación de teclas `<ctrl+c>`.

3. Respecto al funcionamiento de la función principal, el mismo se encarga de contabilizar los pasos que ha dado el conejo hasta cierto punto, además de calcular los sucesores y sus respectivos costos para una posición [X, Y] donde se encuentre el conejo.
4. Una vez que se realiza el cálculo de la función de costo para cada posible sucesor, el algoritmo se encarga de elegir el de menor costo, desplazando el conejo hacia la dirección que represente dicho sucesor, verificando si se llegó a una meta y de ser así, se disminuyen las zanahorias faltantes (para que el conejo se de por satisfecho).

Detalle de la función de costo $f(n)$

Respecto a esta función, se debe tener claro que la misma está condicionada por un costo acumulado $g(n)$ y un heurístico $h(n)$ que indica el costo de una cierta posición hacia la meta.

Entonces, la función de costo para desplazar el conejo a una posición determinada, está dada por:

$$f(n) = g(n) + h(n)$$

Costo acumulado

Tal como se menciona en el instructivo del proyecto, el acumulado hasta un punto específico en la ejecución del algoritmo será simplemente la cantidad de pasos que ha dado el conejo. Dejando así que dicho costo sea simplemente un contador de "pasos actuales", y por ende no se diseña ninguna función para representarle.

La fórmula de este costo estaría dada por:

$$g(n) = \text{Cantidad de Pasos Actuales}$$

Costo del heurístico

Consideraciones del Instructivo

Primero que todo, antes de abarcar el diseño del heurístico, se considera importante considerar las siguientes instrucciones dadas en el instructivo del proyecto y que tienen un impacto importante en el diseño del heurístico:

- El ambiente no es completamente observable.
- El conejo tendrá un rango de visión.
- El diseño del heurístico no debe incluir "memoria" que haga el ambiente implícitamente observable.
- Se indicará al inicio del programa cuántas zanahorias en total debe buscar el conejo para terminar su labor, y así darse por satisfecho después de comerse esa cantidad.

Diseño de la función

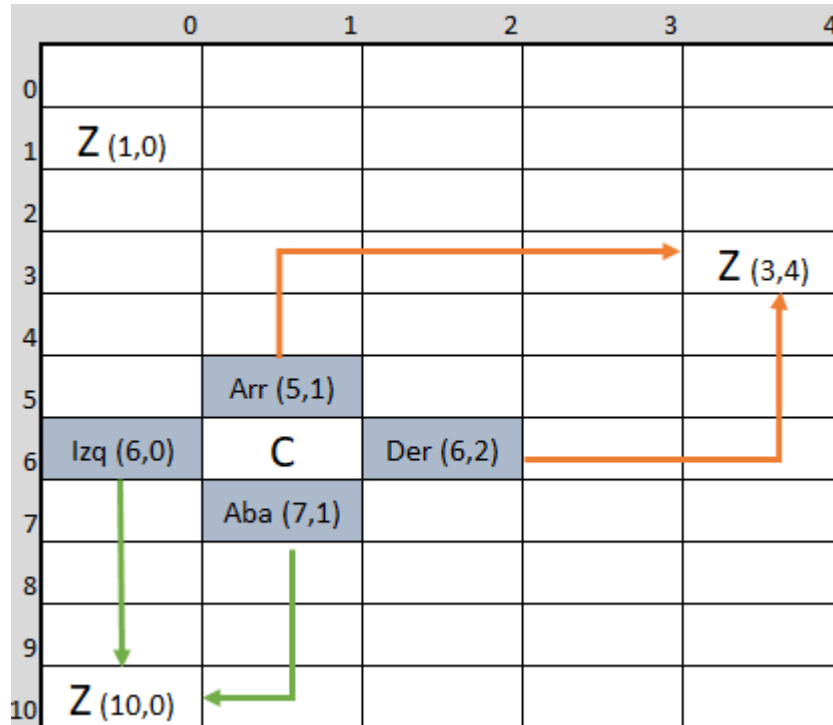
Una vez que se tienen claras las consideraciones del instructivo, se puede proceder con el diseño del heurístico. Mismo que se diseña en conjunto con el grupo de trabajo, donde se concluyen las siguientes consideraciones que fueron implementadas en la función.

1. Aplicación de un costo según la distancia lineal a las zanahorias

En este caso, el costo que se le aplica a un sucesor es básicamente la distancia más corta hacia una zanahoria que se encuentre dentro de su rango de visión. A continuación se muestra el llamado a la función que se encuentra en el código para el cálculo del heurístico.

```
costo_sucesores = castigar_distancia(sucesores, zanahorias, pasos_actuales)
```

Para poner un ejemplo de este cálculo se presenta la siguiente imagen:



Entonces como se puede ver, el conejo podría desplazarse a cuatro posibles direcciones, que serían las que están de color azul.

```
Izquierda, en la posición [6, 0]
Derecha, en la posición [6, 2]
Arriba, en la posición [5, 1]
Abajo, en la posición [7, 1]
```

Luego, para cada una de estas se determina la zanahoria más cercana y se calcula la distancia. Nótese que las zanahorias más cercanas para cada dirección, se encuentran señaladas por una flecha.

Por último, el cálculo de la distancia que representa el costo que se le va agregar a cada dirección se muestra a continuación:

```
Izquierda [6, 0] -> Zanahoria [10, 0] -> Costo = 4
Derecha [6, 2] -> Zanahoria [3, 4] -> Costo = 5
Arriba [5, 1] -> Zanahoria [3, 4] -> Costo = 5
Abajo [7, 1] -> Zanahoria [10, 0] -> Costo = 4
```

2. Aplicación de un costo según la región que tenga más zanahorias

En este caso, el costo que se le aplica a un sucesor es básicamente si su dirección cuenta con menos zanahorias que la dirección opuesta. Es decir, que solamente se aplicará un costo a dos direcciones (Izquierda/Derecha y Arriba/Abajo).

Además, se debe tener claro que este costo solamente se tomará en cuenta si el numero de zanahorias agrupadas cumplen con la cantidad que el conejo ocupa comerse para darse por satisfecho. Esto último se decidió, debido a que la búsqueda no era tan efectiva si se le daba prioridad a la región con más zanahorias.

A continuación se muestra el llamado a la función que se encuentra en el código para el cálculo del heurístico.

```
costo_sucesores = castigar_emisferios(matriz_visible, costo_sucesores,
                                     pos_actual, cant_zanahorias)
```

Para tener claro un ejemplo del cálculo, se debe contemplar cómo se está realizando la comparación entre dos regiones. Esto se logra por medio de un split de la matriz, según la posición que ocupa el conejo.

Entonces para penalizar la dirección Izquierda o Derecha se hace un split vertical, tal como se muestra en la siguiente imagen.

	0	1	2	3	4
0					
1	Z (1,0)				
2					
3					
4				Z (3,4)	
5					
6					
7		C			
8					
9					
10	Z (10,0)				

Luego, para penalizar la dirección Arriba o Abajo se hace un split horizontal, tal como se muestra en la siguiente imagen.

	0	1	2	3	4
0					
1	Z (1,0)				
2					
3					Z (3,4)
4					
5					

6		C			
7					
8					
9					
10	Z (10,0)				

Finalmente, para cada uno de estos se calcula cuál región tiene más zanahorias, y se castiga la dirección opuesta.

NOTA: El costo agregado a la dirección con menos zanahorias es de **3**.

3. Aplicación de un costo si el sucesor va a un espacio desconocido


Este costo es agregado al sucesor que se dirija hacia una posición fuera del tablero de juego.

A continuación se muestra el llamado a la función que se encuentra en el código para el cálculo del heurístico.

```
costo_sucesores = castigar_esp_desconocido(costo_sucesores, forma_matriz)
```

Un ejemplo de esto es el costo que se le agrega a la dirección 'Abajo', en el caso de la siguiente imagen:

	0	1	2	3	4
0					
1	Z (1,0)				
2					
3					Z (3,4)
4					
5					
6					
7					
8					
9					
10	Z (10,0)		C		


Penalizar Abajo

NOTA: El costo agregado para esta penalización es de **100**.

4. Aplicación de un costo a la dirección del padre, si en la misma no existía zanahoria

Este costo es agregado a la dirección de la que provenía el conejo si en esta no existía una zanahoria. Lo que se pretende, es evitar que el conejo quiera devolverse a una posición ya explorada, donde no existía ninguna meta.

A continuación se muestra el llamado a la función que se encuentra en el código para el cálculo del heurístico.

```
costo_sucesores = castigar_direccion_padre(costo_sucesores,
                                           direccion_vieja)
```

NOTA: El costo agregado para esta penalización es de **10**.

En resumen, la función del heurístico podría resumirse en:

```
h(n) = castigar_distancia(n) + castigar_emisferios(n) + castigar_esp_desconocido(n) +
castigar_direccion_padre(n)
```

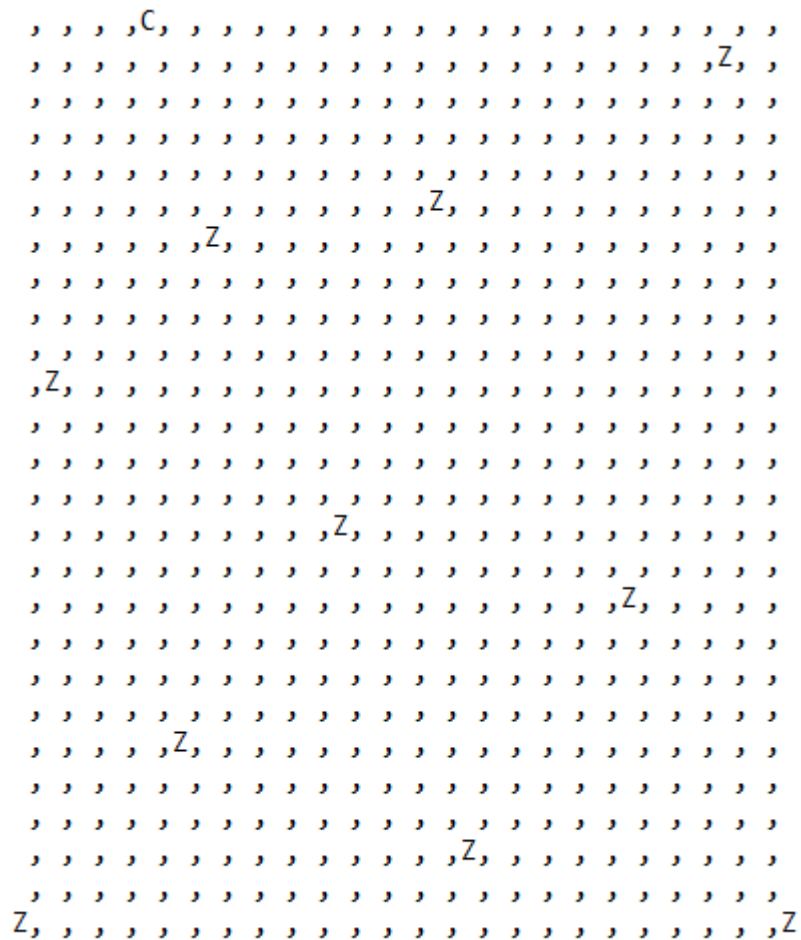
Análisis de resultados

Para el análisis de resultados, se decide mostrar la ejecución de dos tableros de dimensión 25x25 y 10 zanahorias, el cual se utiliza en 20 pruebas con diferentes rangos de visión y cantidad de zanahorias que ocupa el conejo para darse por satisfecho.

A continuación, se muestran dichos tableros y el comportamiento que tienen en función de la cantidad de pasos que tuvo que dar el conejo para alcanzar la meta:

Prueba Número 1

- Tablero utilizado

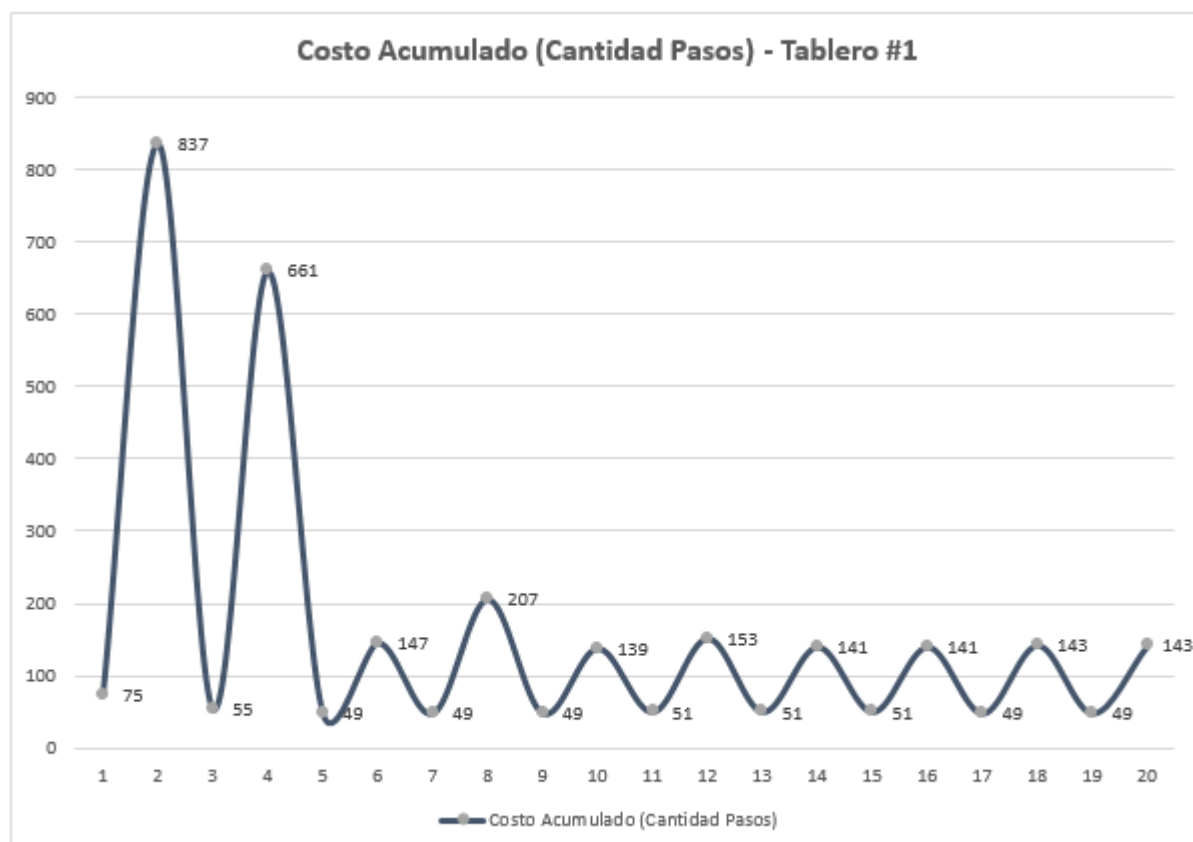


- Resumen de datos

Tablero #1			
# Prueba	Rango Visión	Cantidad Zanahorias	Costo Acumulado (Cantidad Pasos)
1	5	5	75
2	5	10	837
3	7	5	55
4	7	10	661
5	9	5	49
6	9	10	147
7	11	5	49
8	11	10	207
9	13	5	49
10	13	10	139
11	15	5	51
12	15	10	153
13	17	5	51
14	17	10	141
15	19	5	51
16	19	10	141
17	21	5	49
18	21	10	143
19	23	5	49
20	23	10	143

- Gráfico de comportamiento

Tal como se puede apreciar en el siguiente gráfico, el comportamiento en función de costo disminuye conforme el rango de visión aumenta y también cuando la cantidad de zanahorias para darse por satisfecho disminuyen.



Prueba Número 2

- Tablero utilizado

A 20x20 grid of blue dots. The following letters are placed at specific intersections:

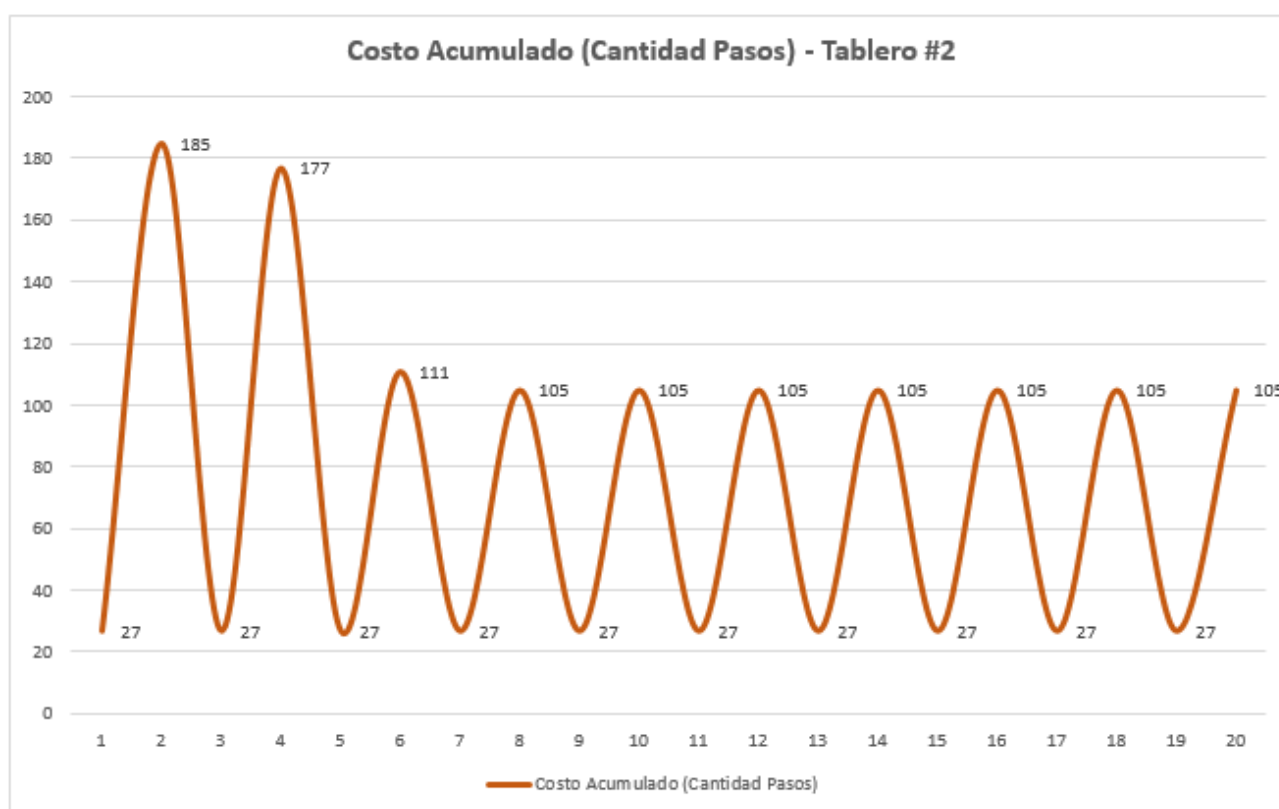
- Row 1: 'c' at column 4.
- Row 4: 'z' at column 16.
- Row 5: 'z' at column 2.
- Row 6: 'z' at column 14.
- Row 7: 'z' at column 12 and 'z' at column 16.
- Row 8: 'z' at column 2.
- Row 10: 'z' at column 14.
- Row 11: 'z' at column 16.
- Row 12: 'z' at column 12.
- Row 13: 'z' at column 16.
- Row 14: 'z' at column 18.
- Row 15: 'z' at column 12.
- Row 16: 'z' at column 16.
- Row 17: 'z' at column 14.
- Row 18: 'z' at column 12.
- Row 19: 'z' at column 16.
- Row 20: 'z' at column 12.

- Resumen de datos

Tablero #2			
# Prueba	Rango Visión	Cantidad Zanahorias	Costo Acumulado (Cantidad Pasos)
1	5	5	27
2	5	10	185
3	7	5	27
4	7	10	177
5	9	5	27
6	9	10	111
7	11	5	27
8	11	10	105
9	13	5	27
10	13	10	105
11	15	5	27
12	15	10	105
13	17	5	27
14	17	10	105
15	19	5	27
16	19	10	105
17	21	5	27
18	21	10	105
19	23	5	27
20	23	10	105

- Gráfico de comportamiento

En esta segunda prueba, se utiliza un tablero 'más sencillo', donde se obtiene un comportamiento muy similar, donde la función de costo disminuye conforme el rango de visión aumenta y también cuando la cantidad de zanahorias para darse por satisfecho disminuyen.



Algoritmos Genéticos

El objetivo principal fue desarrollar un algoritmo genético que optimice la colocación de señales direccionales para que el conejo recorra el tablero.

Detalles de implementación

Previo a la implementación del algoritmo genético, fue necesario tomar algunas decisiones con respecto al comportamiento del mismo. Lo anterior dado que el enunciado del proyecto no podría ni debería cubrir todos los detalles relacionados al algoritmo. Dichos detalles se enumeran a continuación:

1. **Importante:** se decidió que durante el recorrido del conejo, al igual que las zanahorias, las flechas desaparecen del tablero una vez que el conejo llega a la celda que contiene dicha flecha. Esto no se ve reflejado en los archivos de salida finales ni en la consola durante la ejecución, pero debe tenerse presente al interpretar la solución.
 - La motivación para esto radica en evitar los ciclos infinitos durante el recorrido del conejo.
2. Para la mutación, primero se obtiene un número aleatorio de celda, si la celda está vacía, se inserta una flecha aleatoria. Si la celda ya tiene una flecha, se genera un aleatorio para borrar o girar la flecha. Si contiene una zanahoria o al conejo, no sucede nada.
3. Parte de la función de aptitud consiste en contar flechas que están apuntando a una zanahoria, entonces por cada zanahoria se cuenta una única dirección por cada una de las cuatro direcciones disponibles.
4. Se da prioridad a brindar una solución con menor cantidad de flechas por encima de menor cantidad de pasos, esto pues no hay un criterio objetivo para diferir entre pasos o flechas. Pero si se observó que entre menor cantidad de flechas, más fácil resulta la interpretación de la solución final.
5. El recorrido del camino del conejo, según las flechas, se da por terminado una vez que el conejo ha recolectado todas las zanahorias o se ha salido del tablero.
6. A pesar de que el tablero utilizado se considera una matriz, para apegarse al estándar de algoritmos genéticos, este se transforma a un arreglo unidimensional en ocasiones, según la necesidad. Para el recorrido del tablero, se hace con forma de matriz, pero para el cruce se utiliza la representación como arreglo unidimensional. Esto pues así se facilita el corte en cualquier celda, y no necesariamente por filas o columnas.

Detalle de la función de aptitud

La función de aptitud es el componente principal que define los resultados obtenidos con el algoritmo genético. A continuación, se mencionan los aspectos tomados en cuenta para su definición.

En términos generales, la función de aptitud recorre el camino que debería seguir el conejo con indicación de las "flechas", durante dicho recorrido se contabilizan los aspectos considerados importantes. También se toman en cuenta algunos aspectos que no necesitan del recorrido del conejo, por ejemplo la cantidad de flechas que apuntan a una zanahoria.

Como detalle importante, se toma en cuenta el tamaño del tablero para escalar algunos de los pesos, esto con el fin de que sin importar el tamaño del tablero, algunos pesos sigan siendo de más valor que otros. Esto se logra mediante la variable `_scalar` obtenida al multiplicar la cantidad de filas del tablero por la cantidad de columnas. De igual forma para dar un poco de consistencia a los valores de los pesos, dicho

escalar su "redondea hacia arriba" al múltiplo más cercano de un determinado número mediante la función `round_up(n, multiplo)`.

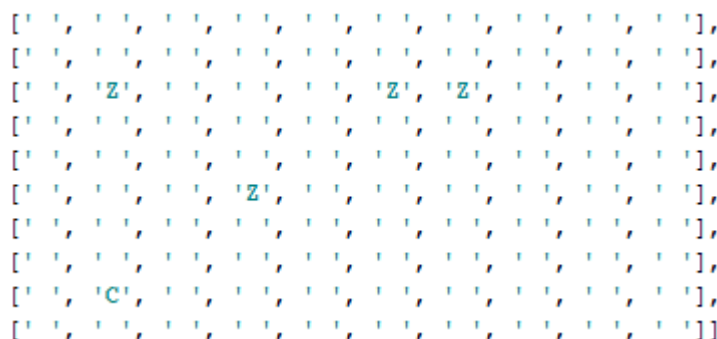
1. **Zanahorias recogidas.** Se contabilizan cuantas zanahorias logró recoger el conejo antes de caer. O en su defecto si recogió todas las zanahorias del tablero. Tiene el mayor peso positivo en la aptitud.
2. **Pasos.** Conteo de los pasos o casillas que ha recorrido el conejo. Para evitar que la aptitud sea mayor para un conejo que se sale inmediatamente del tablero, los pasos son contados como positivos si el conejo se sale y negativos si el conejo logra conseguir todas las zanahorias.
3. **Flechas encontradas.** Conteo de las flechas encontradas o utilizadas durante el recorrido del conejo. Estas disminuyen la aptitud, para minimizar la cantidad de flechas usadas.
4. **Flechas no encontradas.** Conteo de flechas presentes en el tablero pero que no fueron recorridas por el conejo. Estas disminuyen aún más, que las encontradas, la aptitud de un tablero.
5. **Flechas que apuntan a zanahorias.** Conteo de flechas que apuntan a una zanahoria, tienen un peso positivo que contrarresta la penalización para flechas encontradas, pero no es suficientemente alto para contrarrestar la penalización por flechas no encontradas.
6. **Giros de 180°.** Se penalizan altamente los cambios de dirección opuesta totalmente. Por la implementación estos cambios no generan bucles infinitos, pero no tienen sentido a menos que sean después de haber recogido una zanahoria, en cuyo caso podría ser de utilidad. Por lo anterior, los giros de 180° luego de recoger una zanahoria, no se penalizan.
7. **Mejor zanahoria inicial.** Se otorga un peso positivo extra al tablero, si la primera zanahoria recogida es la más cercana (utilizando la distancia lineal desde el conejo a cada zanahoria). Esto con el fin de que al menos la primera zanahoria recogida genere la menor cantidad de pasos posibles, en caso de ser posible.

Tasa de mutación

La tasa de mutación es un número entre 0 y 100, que define el porcentaje o probabilidad con que mutan los resultados de un cruce. El algoritmo de una mutación se aborda en el punto 2 de los detalles de implementación.

Resultados según tasa de mutación

Para efectos de las pruebas del algoritmo se utilizaron porcentajes de 40% y 80% de mutación, para la comparación del efecto que tiene duplicar la mutación de genes. Todos los análisis se hicieron sobre el mismo tablero de juego. Representado con la siguiente imagen.



Efecto en puntajes de aptitud

Para obtener los resultados descritos en esta sección, se debe importar la función

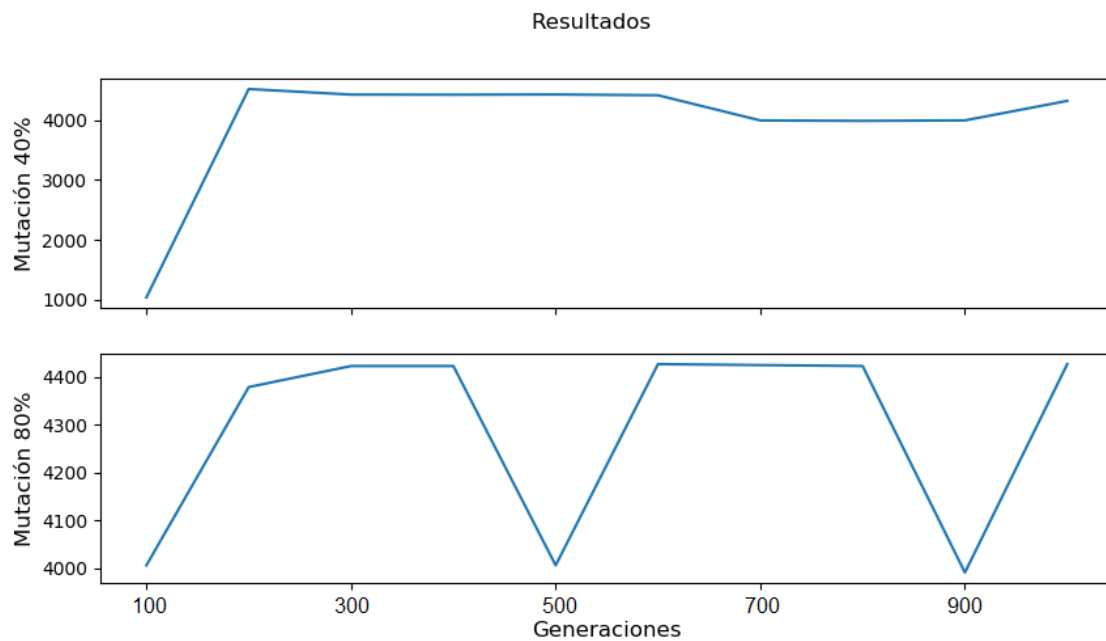
```
mutation_chance_effect_on_scores(semilla_del_aleatorio)
```

```
from tec.ic.ia.pc2.model.ga_analysis import mutation_chance_effect_on_scores
```

Como **métrica** de evaluación, se calcula la diferencia resultante al restarle, a la suma de los puntajes obtenidos para mutación de 40%, la suma de los puntajes obtenidos para mutación de 80%.

1. Se establece la semilla del random = 5.

```
mutation_chance_effect_on_scores(5)
```



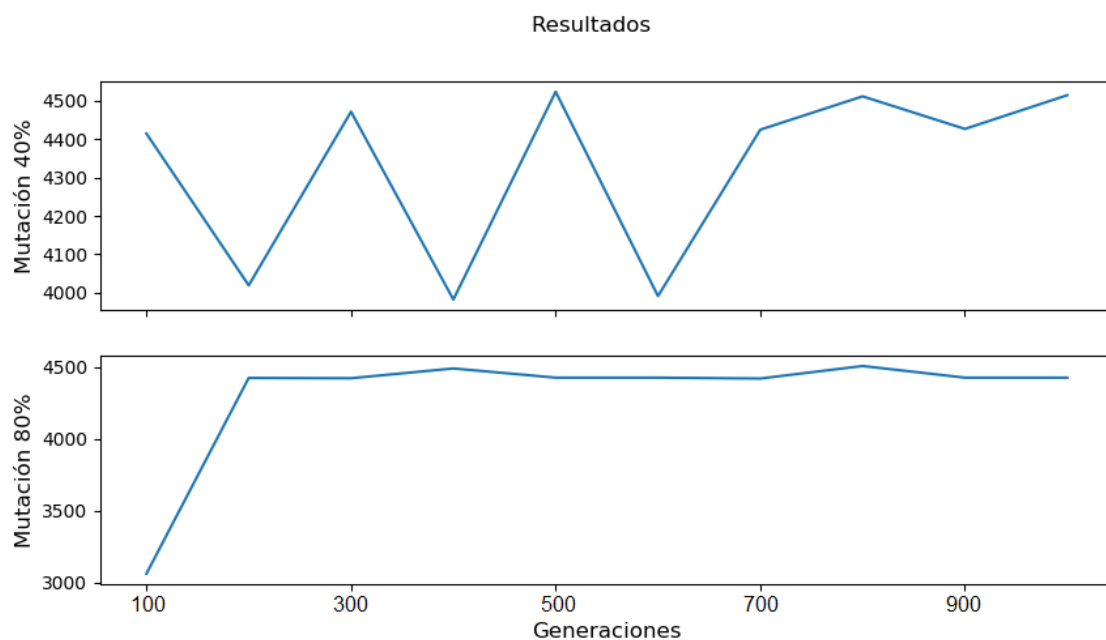
Diferencia obtenida: -3393

Diferencia obtenida sin contar la peor solución: -439

Según la métrica anterior, un 80% de probabilidad de mutación genera puntajes de aptitud más altos por 439 puntos. Hay que destacar que un 40% no fue suficiente para obtener una solución que se coma todas las zanahorias en 100 generaciones.

2. Se establece la semilla del random = 1996.

```
mutation_chance_effect_on_scores(1996)
```

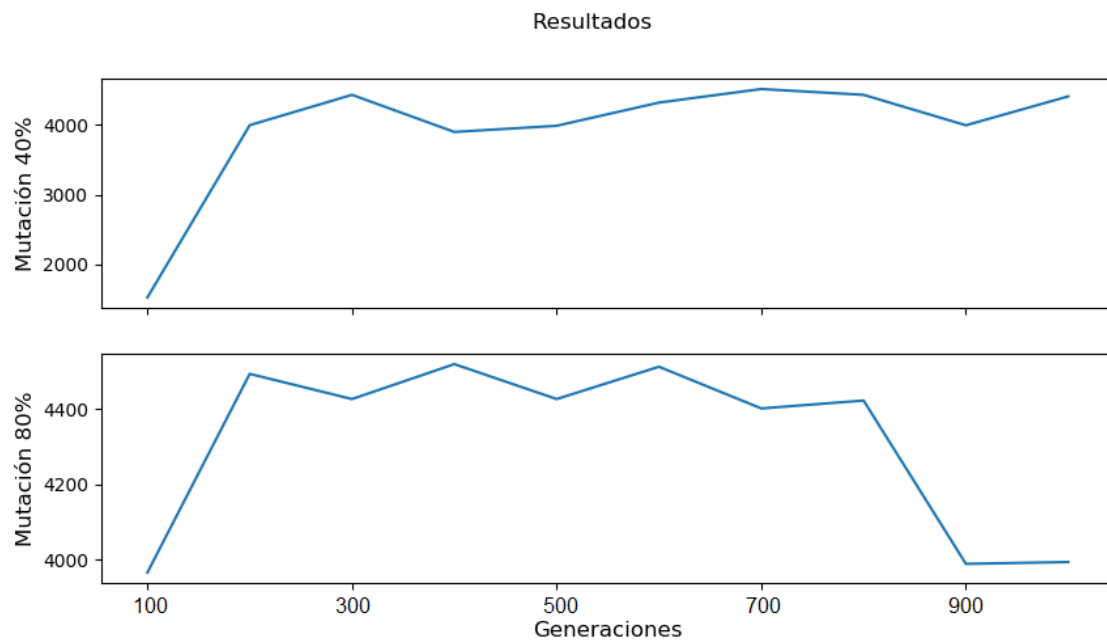


Diferencia obtenida: 243

Según la métrica anterior, un 40% de probabilidad de mutación genera puntajes de aptitud más altos por 243 puntos.

3. Se establece la semilla del random = 2011.

```
mutation_chance_effect_on_scores(2011)
```



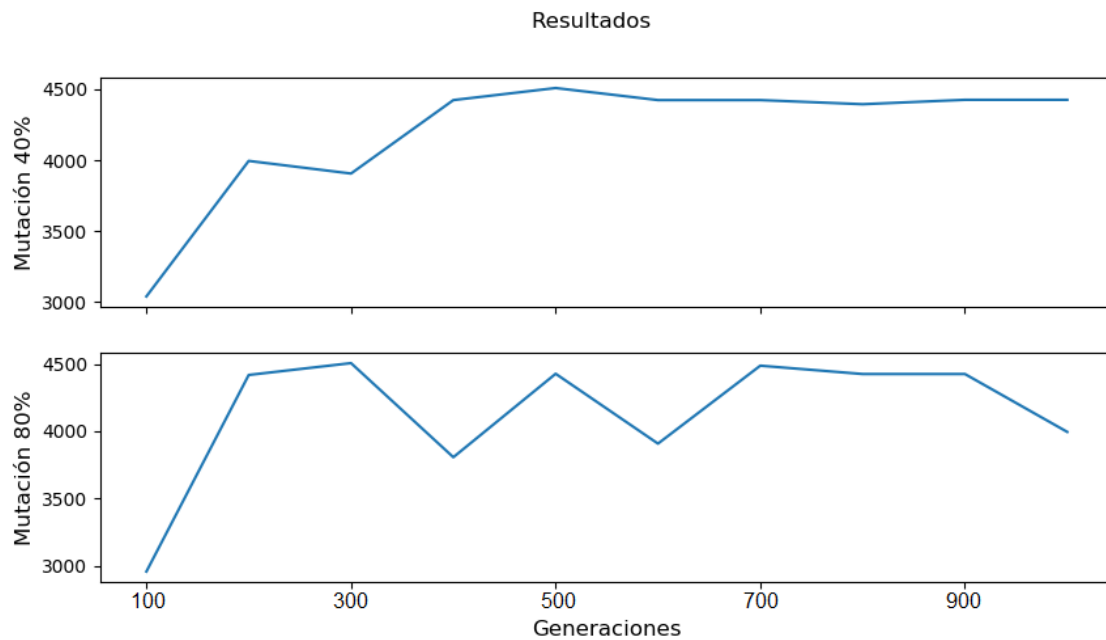
Diferencia obtenida: -3674

Diferencia obtenida sin contar la peor solución: -1238

Según la métrica anterior, un 80% de probabilidad de mutación genera puntajes de aptitud más altos por 1238 puntos. Hay que destacar que nuevamente, un 40% de probabilidad de mutación no fue suficiente para brindar una solución que se coma todas las zanahorias en 100 generaciones.

4. Se establece la semilla del random = 2018.

```
mutation_chance_effect_on_scores(2018)
```

Diferencia obtenida: 607

Según la métrica anterior, un 40% de probabilidad de mutación genera puntajes de aptitud más altos por 607 puntos.

Conclusiones

1. Según las pruebas realizadas, un porcentaje más bajo de mutación puede significar que el algoritmo no será capaz de brindar un acomodo de flechas suficientemente bueno para comerse todas las zanahorias.
2. Según la definición de los pesos que afectan el puntaje de aptitud, es difícil definir si una tasa de mutación alta se mejor que una baja. Pero se logra rescatar que ante una tasa de mutación alta, la creación de genes es mayor, lo cual significa un incremento considerable en la cantidad de recursos del sistema a utilizar. Por lo tanto, existe una relación entre cantidad de generaciones y tasa de mutación, que debe balancearse con respecto a la cantidad de recursos por gastar.

Efecto en la velocidad para encontrar la solución

Para obtener los resultados descritos en esta sección, se debe importar la función

```
mutation_chance_effect_on_speed(semilla_del_aleatorio)
```

```
from tec.ic.ia.pc2.model.ga_analysis import mutation_chance_effect_on_speed
```

Se realizaron 10 ejecuciones para cada porcentaje de mutación (40% y 80%). Los resultados obtenidos son el mejor puntaje obtenido y el número de generación en el que se obtuvo.

Para todas las ejecuciones se utilizaron los mismos parámetros, a excepción del porcentaje de mutación que varía entre ambas tablas.

```
initial_direction='derecha', individuals=15, max_generations=550, cross_type=1
```

Función ejecutada

```
mutation_chance_effect_on_speed(2018)
```

Resultados:

Mutación	40%		Mutación	80%
Puntaje	N° Generación		Puntaje	N° Generación
4427	40		4522	52
4427	77		3991	70
4522	79		4507	85
4423	167		3980	160
4423	226		3993	179
3989	227		4522	236
3980	254		4370	295
4318	390		3982	305
4507	428		3996	335
4489	482		4427	393

Conclusiones

1. Las pruebas realizadas muestran puntajes relativamente similares, por lo que resulta útil para la comparación entre el número de generación donde se origina el resultado.
2. En 2 ocasiones, la mutación con 40% de probabilidad dio como resultado soluciones en generaciones mayores a la 400. Por lo que aparenta ser que un mayor porcentaje de mutación puede disminuir en general la cantidad de generaciones necesarias.
3. Para soluciones obtenidas en generaciones menores a la 400, los resultados son bastante uniformes, por lo que aparenta ser que la diferencia radica en lo mencionado en el punto 2 únicamente.

Política de cruce

Aspectos sobre el algoritmo de cruce se encuentran en el punto 6 de la sección de detalles de implementación. Se implementaron dos políticas de cruce diferentes.

1. **Cruce por corte en un punto.** Cada arreglo padre se divide en dos, el punto de división es el mismo para ambos, y se determina mediante un número aleatorio sobre las posibles casillas. Luego se intercambian una parte de un padre con su correspondiente parte del otro padre.

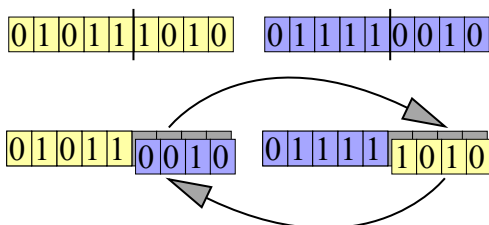


Imagen recuperada de: [https://upload.wikimedia.org/wikipedia/c](https://upload.wikimedia.org/wikipedia/commons/d/dd/Computational.science.Genetic.algorithm.Crossover.One.Point.svg)

[ommons/d/dd/Computational.science.Genetic.algorithm.Crossover.One.Point.svg](https://upload.wikimedia.org/wikipedia/commons/d/dd/Computational.science.Genetic.algorithm.Crossover.One.Point.svg)

2. **Cruce por corte en dos puntos.** Cada arreglo padre se divide en tres, los dos puntos de división son los mismos para ambos, y se determinan mediante dos números aleatorios sobre las posibles casillas. Luego se intercambian la parte central de un padre con su correspondiente parte del otro padre.

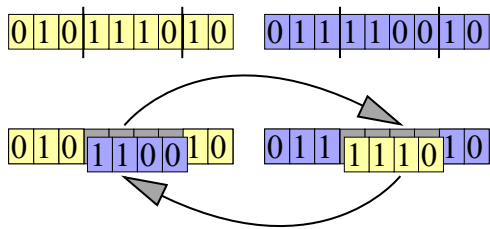
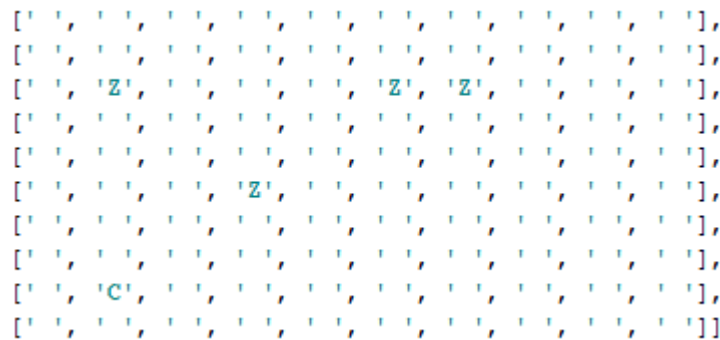


Imagen recuperada de: [https://upload.wikimedia.org/wikipedia/c](https://upload.wikimedia.org/wikipedia/commons/4/47/Computational.science.Genetic.algorithm.Crossover.Two.Point.svg)

[ommons/4/47/Computational.science.Genetic.algorithm.Crossover.Two.Point.svg](https://upload.wikimedia.org/wikipedia/commons/4/47/Computational.science.Genetic.algorithm.Crossover.Two.Point.svg)

Resultados según política de cruce

Para efectos de las pruebas del algoritmo se utilizaron las políticas de cruce con corte en uno y dos puntos. Todos los análisis se hicieron sobre el mismo tablero de juego. Representado con la siguiente imagen.



Efecto en puntajes de aptitud

Para obtener los resultados descritos en esta sección, se debe importar la función

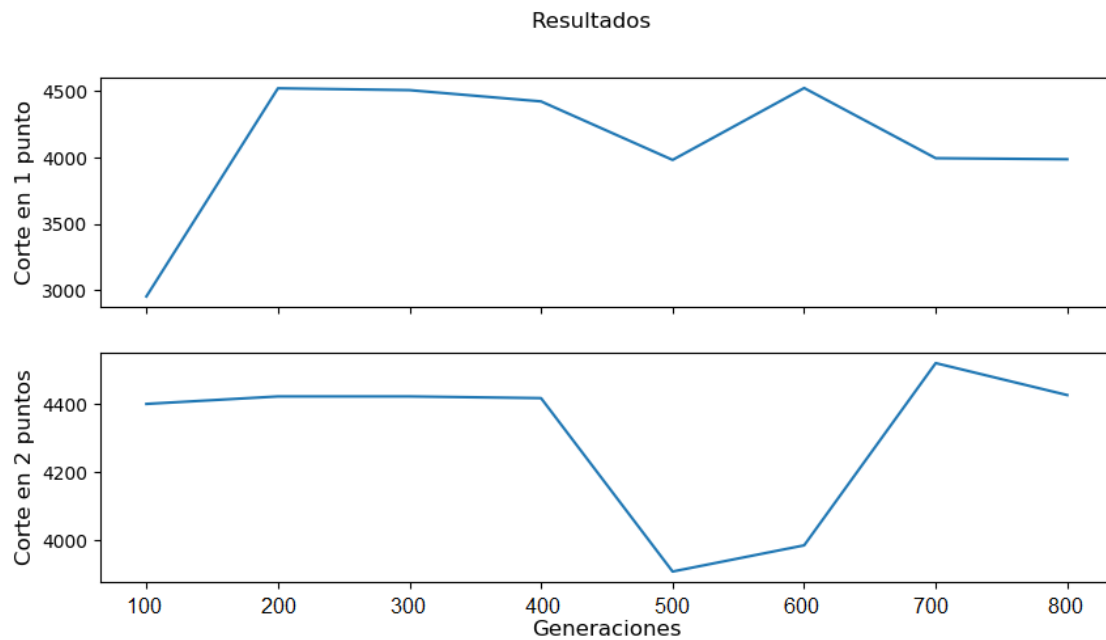
```
cross_policy_effect_on_scores(semilla_del_aleatorio)
```

```
from tec.ic.ia.pc2.model.ga_analysis import cross_policy_effect_on_scores
```

Como **métrica** de evaluación, se calcula la diferencia resultante al restarle, a la suma de los puntajes obtenidos para el cruce con corte en un punto, la suma de los puntajes obtenidos para cruce con corte en dos puntos.

1. Se establece la semilla del random = 5.

```
cross_policy_effect_on_scores(5)
```



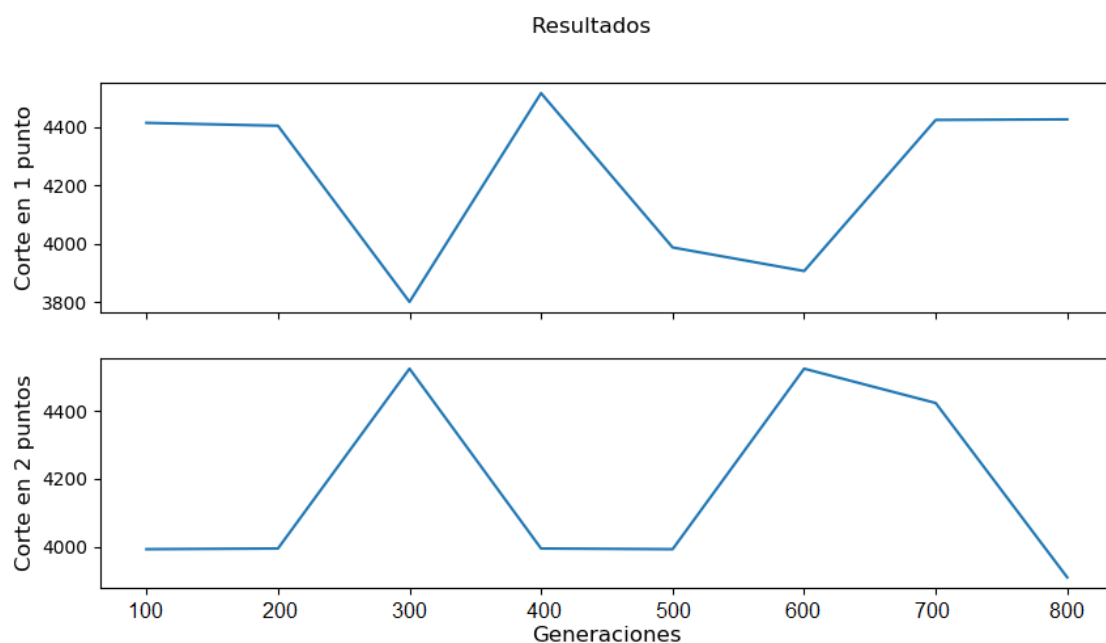
Diferencia obtenida: -1612

Diferencia obtenida sin contar la peor solución: -657

Según la métrica anterior, un corte en dos puntos genera puntajes de aptitud más altos por 657 puntos. Similar a lo que sucedía con la mutación, el corte en un punto no fue capaz de recolectar todas las zanahorias para 1 de las 8 ejecuciones realizadas. Esto podría ser producto del azar o estar relacionado a la política de cruce directamente.

2. Se establece la semilla del random = 1996.

```
cross_policy_effect_on_scores(1996)
```

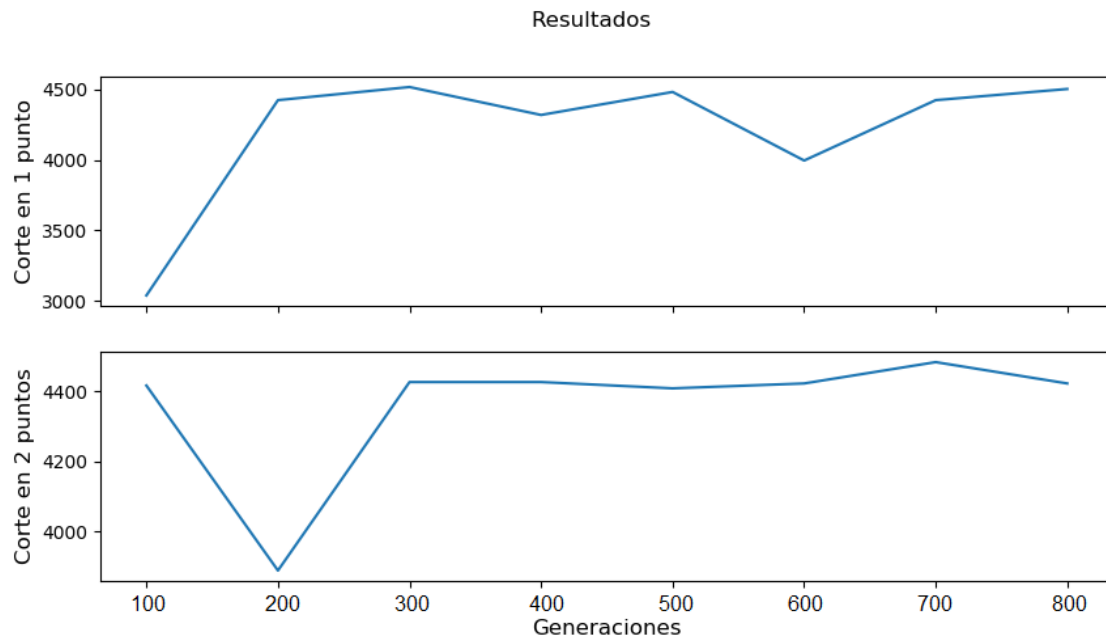


Diferencia obtenida: 525

Según la métrica anterior, un corte en un solo punto genera puntajes de aptitud más altos por 525 puntos.

3. Se establece la semilla del random = 2011.

```
cross_policy_effect_on_scores(2011)
```



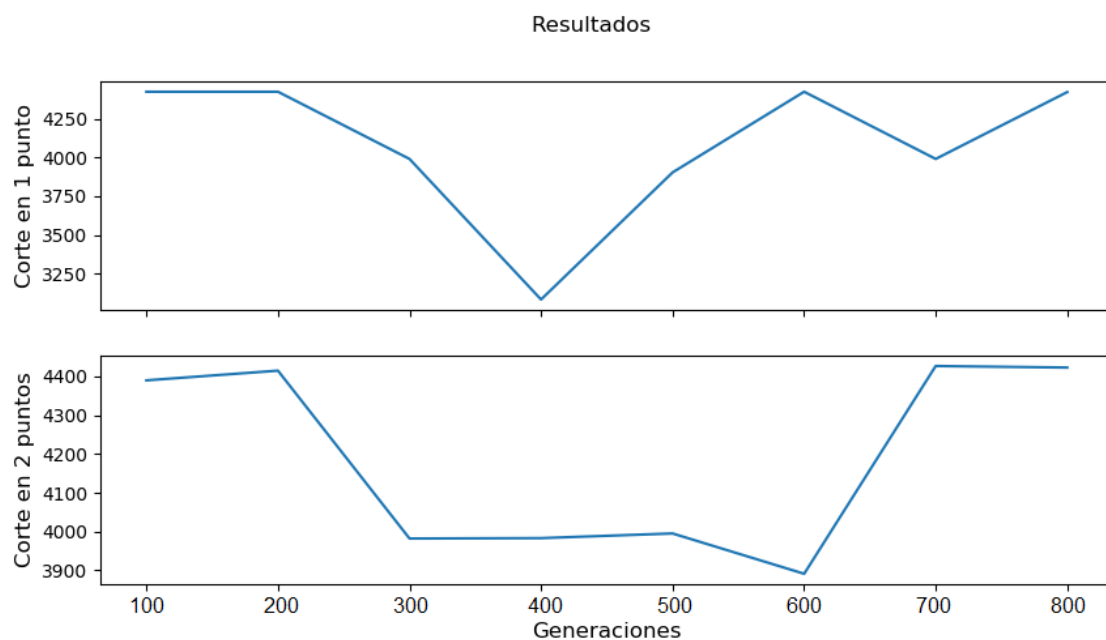
Diferencia obtenida: -1201

Diferencia obtenida sin contar la peor solución: -352

Según la métrica anterior, un corte en dos puntos genera puntajes de aptitud más altos por 352 puntos. Hay que destacar que nuevamente, un corte en un solo punto no fue suficiente para brindar una solución que se coma todas las zanahorias en 100 generaciones.

4. Se establece la semilla del random = 2018.

```
cross_policy_effect_on_scores(2018)
```



Diferencia obtenida: -839

Según la métrica anterior, un corte en dos puntos genera un resultado de aptitud más alto por 839 puntos. Nuevamente el corte en un solo punto no generó una solución que se coma todas las zanahorias, pero esta vez a nivel de 400 generaciones.

Conclusiones

1. Según las pruebas realizadas, un corte en un solo punto en el arreglo del gen puede significar que el algoritmo no será capaz de brindar un acomodo de flechas suficientemente bueno para comerse todas las zanahorias.
2. Parece ser que en esta ocasión a diferencia de los porcentajes de mutación, si existe una ventaja clara en usar la segunda política de cruce definida por sobre la primera. El corte en dos puntos del arreglo que representa el gen, parece generar mejores resultados que el corte en un único punto.

Efecto en la velocidad para encontrar la solución

Para obtener los resultados descritos en esta sección, se debe importar la función

```
cross_policy_effect_on_speed(semilla_del_aleatorio)
```

```
from tec.ic.ia.pc2.model.ga_analysis import cross_policy_effect_on_speed
```

Se realizaron 10 ejecuciones para cada política de cruce. Los resultados obtenidos son el mejor puntaje obtenido y el número de generación en el que se obtuvo.

Para todas las ejecuciones se utilizaron los mismos parámetros, a excepción de la política de cruce que varía entre ambas tablas.

```
initial_direction='derecha', individuals=15, max_generations=550, mutation_chance=50
```

Función ejecutada

```
cross_policy_effect_on_speed(2018)
```

Resultados:

Política de Cruce	Corte en 1 Punto		Política de Cruce	Corte en 2 Puntos
Puntaje	Nº Generación		Puntaje	Nº Generación
4425	27		4427	76
4425	39		4427	133
4425	39		4427	135
4425	140		3908	154
3976	161		4019	161
4425	186		4423	163
4427	203		4522	192
4391	227		4520	228
3910	237		4427	295
3984	248		4425	329

Conclusiones

1. Las pruebas realizadas muestran puntajes relativamente similares, por lo que resulta útil para la comparación entre el número de generación donde se origina el resultado.
2. El corte en 1 punto dio la solución final en generaciones muy bajas, osea pocas iteraciones, en 3 ocasiones. Las 3 ocasiones están por debajo de la generación 50. Mientras que la única vez que el cruce en 2 puntos dio una solución en una generación menor a 100, apenas logró darla en la 76. Viendo esto, parece que la política de cruce con corte en 1 punto parece alcanzar la solución en pocas generaciones con mayor frecuencia.
3. Hay que destacar que el corte en un punto generó puntajes de aptitud menores a 4000 en 3 ocasiones, aunque claramente cercanos, esto puede deberse al peso relacionado a conseguir la primer zanahoria más cercana, por lo que podría decirse que el cruce en dos puntos tiende a conseguir la mejor primer zanahoria con más frecuencia, lo cual mejora el puntaje de la solución.

Acerca de

Integrantes del proyecto:

Nombre	Carné
Brandon Dinarte Chavarría	2015088894
Armando López Cordero	2015125414
Julian Salinas Rojas	2015114132

Estudiantes de Ingeniería en Computación del Instituto Tecnológico de Costa Rica.