|  |  |  |
| --- | --- | --- |
| **ipn** | **INSTITUTO POLITÉCNICO NACIONAL**  **ESCUELA SUPERIOR DE CÓMPUTO** |  |

**Análisis de Algoritmos**

**“Proyecto – Algoritmos Greedy vs Fuerza Bruta”**

Resumen

El resumen debe ser preciso de todo lo que trata el reporte completo. Su función es indicar los contenidos del reporte para que el lector pueda ver si vale la pena leerlo completo o no.

**Por:**

**González Barrios Alan Ernesto**

**Romero Gamarra Joel Mauricio**

**Zavala Pérez René**

Profesor:

FRANCO MARTÍNEZ EDGARDO ADRIAN

Diciembre 2017

**Índice**

Contenido

[Introducción: 1](#_Toc500873674)

[Análisis Teórico: 6](#_Toc500873675)

[Software (librerías, paquetes, herramientas): 12](#_Toc500873676)

[Procedimiento: 13](#_Toc500873677)

[Resultados 13](#_Toc500873678)

[Conclusiones: 17](#_Toc500873679)

[Referencias: 18](#_Toc500873680)

[Código 18](#_Toc500873681)

# Introducción:

Los algoritmos ávidos o voraces (Algoritmos Greedy) son algoritmos que toman decisiones de corto alcance, basadas en información inmediatamente disponible. Este tipo de algoritmos suele ser bastante rápidos para encontrar una solución, que, por cierto, la mayoría de las veces son problemas de optimización (ya sea minimizar un costo o maximizar un beneficio). [1]

Estos algoritmos tienen una serie de características o elementos que lo conforman:

1. **Conjunto de candidatos (C):** Son las entradas del problema.
2. **Función solución:** Comprueba si el subconjunto actual de candidatos elegidos forma o no una solución (sin importar si es óptima o no).
3. **Función de selección:** Indica cual es el elemento más prometedor para completar la solución. Cada elemento puede solamente ser aceptado o rechazado.
4. **Función de factibilidad:** Indica si a partir de un conjunto se puede llegar a una solución.
5. **Función objetivo:** Es la que se quiere maximizar o minimizar.

Lo que hace a los algoritmos Greedy bastante rápidos para hallar la solución es que, en cada paso, van escogiendo la solución óptima, para al final combinar las soluciones, y como todas esas pequeñas soluciones fueron óptimas, la respuesta final al problema será óptima, sin embargo, no sabemos si fue la mejor solución. Esto tiene algunas ventajas y desventajas mostradas a continuación:

**Ventajas:**

* Son mucho más rápidos que los algoritmos que encuentran la mejor respuesta ya que resuelven en problema en 1 solo ciclo.
* La programación de los mismos es bastante simple, sin embargo, encontrar la lógica no lo es tanto.
* Al resolver problemas de optimización, tiene aplicaciones como encontrar el camino más corto en un grafo de 1 punto a otro (Telecomunicaciones).
* Si toman una decisión, no se regresan a ver si la decisión tomada fue la mejor, pero sabemos que es óptima (evitan redundancia).
* La solución entregada, es óptima.

**Desventajas:**

* No garantizan que la solución entregada del problema sea la mejor solución posible.
* No todos los problemas pueden ser resueltos por un algoritmo Greedy (solo aquellos que deriven de optimización).
* Al no obtener la mejor solución, se debe comprobar al final que la respuesta entregada fue óptima.

Como podemos ver, existen más ventajas para los algoritmos voraces que desventajas, esto indica que siempre que se presente algún problema de maximizar un beneficio o minimizar un costo, un algoritmo voraz es la respuesta debido a su gran velocidad de resolver el problema en un solo ciclo sin regresar a ver si la decisión tomada (que se agrega al conjunto solución), fue la óptima.

Un algoritmo voraz trabaja por etapas, ya que, en cada una de éstas, toma la decisión que le parece más conveniente sin pensar en las repercusiones que esto pudiera tener un futuro, ya que como en ese momento, ese candidato produce un óptimo local y el algoritmo hace la suposición que provocará un óptimo global para dar una buena solución al problema. [1]

En este proyecto, nos enfocamos en 2 problemas que pueden ser resueltos con un algoritmo voraz, en uno de ellos se nos pide maximizar y en el otro, minimizar. Estos 2 problemas son descritos a continuación:

**Scarecrow [2]**

Taso adquiere un terreno muy grande en el que planea sembrar diferentes cosechas en la próxima temporada de crecimiento. Sin embargo, el terreno está lleno de cuervos y Taso tiene miedo de que los cuervos se vayan a comer sus cosechas. Para evitar esto, en la temporada de crecimiento, está planeando poner espantapájaros en diferentes puntos del terreno (que está modelado como de 1 x N m2), algunas partes del terreno son infértiles, por lo tanto, es inútil poner un espantapájaros ahí ya que no se podrá sembrar nada.

Lo que Taso sabe, es que cuando coloca un espantapájaros, cubre el espacio en el que se puso, y el espacio una posición a la derecha y una posición a la izquierda, como se muestra en la Figura 1.



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Resultado de imagen para cosecha | Resultado de imagen para cosecha | Resultado de imagen para cosecha | Resultado de imagen para cosecha | Resultado de imagen para cosecha |

Figura . Ejemplo del área cubierta por un espantapájaros

Sin embargo, Taso desea minimizar el número de espantapájaros que debe poner en su terreno para proteger su cosecha de los cuervos, recordando que, si una parte del terreno es infértil, no tiene caso proteger esa área.

La entrada para este problema (en el caso particular de este proyecto) es:

* La forma del terreno (el usuario arrastra las imágenes de la cosecha (maíz) y de la tierra infértil a su gusto).

Restricciones:

* El terreno tiene una longitud de 1 x N, con N menor o igual a 10 (para apreciar bien la animación).

En las Figura 2, se muestra la pantalla de inicio para el usuario.

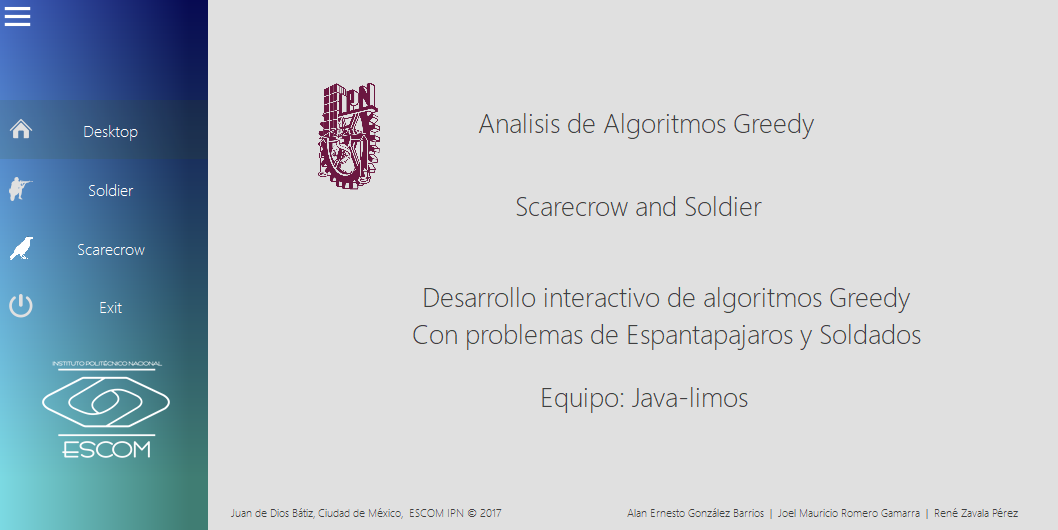


Figura . Pantalla de inicio

A continuación, en las Figuras 3 y 4 se muestra la interfaz en la opción para Scarecrow.

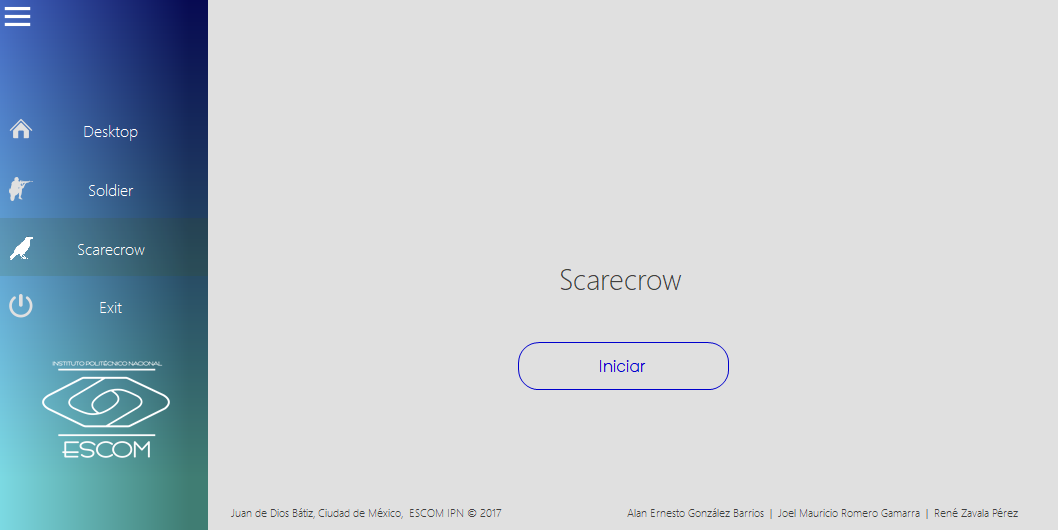


Figura . Interfaz en la opción de Scarecrow



Figura . Interfaz para colocar las cosechas y tierra infértil

Como se puede ver en la parte superior de la Figura 4, las imágenes están puestas para poderlas arrastrar y formar la cadena en los espacios de la parte inferior.

**Bear and Row 01 [3]**

Limak es un oso polar que está jugando un videojuego, hay una fila con N espacios, cada una está vacía u ocupada por un soldado. El objetivo del juego es mover a todos los soldados de la fila hacia la derecha, como se muestra en las Figuras 5 y 6.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |

Figura . Fila de soldados al iniciar el juego

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |

Figura . Fila de soldados al terminar el juego

La única manera de lograrlo es escoger un soldado y darle la orden de que se mueva lo más a la derecha posible, sin embargo, escoger un soldado y darle la orden tarda 1 segundo, además, un soldado tarda 1 segundo en moverse 1 posición, el soldado al que se le dio la orden se detiene inmediatamente si el espacio que sigue ya está ocupado por un soldado.

Como a Limak le gusta mucho este juego, desea jugar el mayor tiempo posible, sin embargo, el no puede dar una orden a otro soldado hasta que el soldado que se está moviendo en ese momento termine de moverse. El objetivo, es decirle a Limak ¿cuántos segundos puede jugar máximo?

La entrada y restricciones para este problema son exactamente las mismas que para el problema de los espantapájaros, con una longitud máxima de la fila de 10 elementos.

En las Figuras 7 y 8, se muestra la interfaz en la opción para Bear and Row 01.

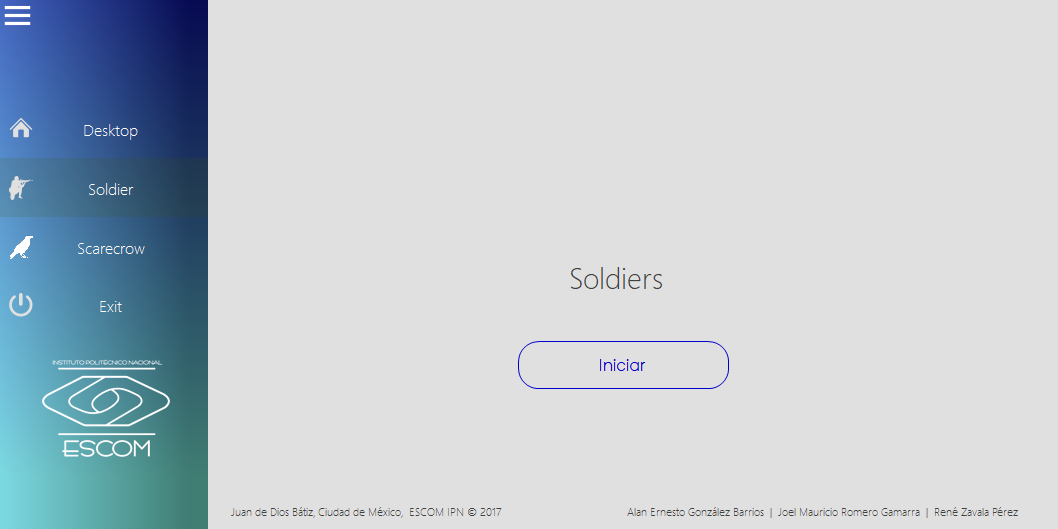


Figura . Interfaz en la opción de Bear and Row 01



Figura . Interfaz para formar a los soldados y obstáculos

Ya que se explicaron los 2 problemas a resolver, en la sección de análisis teórico se explica detalladamente cual sería la solución Greedy y cual la solución con Fuerza Bruta, explicando el porqué de cada uno.

# Análisis Teórico:

Como ya se explicó previamente, ambos algoritmos serán resueltos de 2 maneras distintas, utilizando Fuerza Bruta (algo bastante costoso si fuera una cadena muy larga) y un algoritmo Greedy (que resuelve el problema en un solo ciclo).

**Scarecrow [2]**

* **Fuerza Bruta**

En este caso, el algoritmo de fuerza bruta es bastante sencillo. Como se mencionó en la sección de introducción, el objetivo es proteger todo el terreno pero minimizando el número de espantapájaros a utilizar, sin embargo, eso sería el algoritmo Greedy ya que sabemos que resuelven problemas de optimización, para el caso de resolver el problema por el método de fuerza bruta, lo que haremos es totalmente lo contrario, si proteger todo el terreno, pero utilizando un espantapájaros por cada parte del terreno, sin importar si cubre la parte posterior y anterior, esto maximizará el tiempo en el que son colocados (que es lo que nos interesa analizar en este proyecto).

Para nosotros, cada movimiento de un espantapájaros cuesta 1 segundo, y si tenemos una cadena en la que las 10 posiciones del terreno (longitud del terreno máxima ya definida por nosotros para la correcta visualización) estén ocupadas por la cosecha (en nuestro caso, maíz), entonces habrá 10 espantapájaros, uno sobre cada parte del terreno.

Se comienza recorriendo la cadena de izquierda a derecha, si se encuentra una parte del terreno con cosecha en esa área, se aumentará un espantapájaros, es decir, por cada parte del terreno que tenga maíz (sin importar todo el terreno que cubre un solo espantapájaros) aumentaremos los espantapájaros en 1.

Ya que cada movimiento de un espantapájaros cuesta 1 segundo, entonces en una cadena que se tengan las 10 posiciones ocupadas por cosecha, el tiempo total sería el siguiente (considerando, que cada espantapájaros tarda 1 segundo por cada posición que se mueve a la izquierda y 1 segundo extra en lo que es colocado):

10 + 1 + 9 + 1 + 8 + 1 + 7 + 1 + 6 + 1 + 5 + 1 + 4 + 1 + 3 + 1 + 2 +1 + 1 + 1 = 65 segundos

El total de segundos para el algoritmo por fuerza bruta es de 65 segundos, sin embargo, hay que analizar el tiempo que tardaría el mismo problema con un algoritmo Greedy para tener un punto de comparación y ver si realmente son tan rápidos como se mencionó previamente.

* **Greedy**

En el caso del algoritmo Greedy, haremos uso de lo que ya sabemos.

Como cada espantapájaros cubre la posición justo debajo de él, y las posiciones inmediatamente a la derecha y a la izquierda (un total de 3 posiciones), la resolución del problema sería la siguiente:

Recorremos la cadena de izquierda a derecha y si encontramos alguna parte del terreno que contenga un maíz (sin importar lo que haya después), aumentamos en 1 los espantapájaros necesarios y avanzamos 2 posiciones a la derecha en la cadena (ya que sabemos que un espantapájaros cubre la posición donde está, la de adelante y la de atrás), así que, si encontramos una parte ocupada por maíz, entonces el espantapájaros se colocará en la casilla inmediatamente después (aunque sea tierra infértil, porque cubriría la casilla que ya sabemos que tiene un maíz), y así seguimos recorriendo hasta llegar al final.

Para el mismo caso que en fuerza bruta (es decir, las 10 posiciones ocupadas por maíz), el tiempo que tardaría el algoritmo Greedy sería el siguiente:

9 + 1 + 6 + 1 + 3 + 1 + 1 + 1 = 23 segundos

Los tiempos obtenidos en el algoritmo Greedy son los siguientes:

* 9 segundos = Como hay un maíz en la posición [0], ponemos el espantapájaros en la posición inmediatamente después, es decir, la posición [1] para que cubra la posición anterior y la que sigue, o sea que, un espantapájaros en la posición [1] cubre las posiciones: [0], [1] y [2].
* 6 segundos = Como hay otro maíz en la posición [3], se sigue la misma lógica y el espantapájaros se pone en la posición [4] para cubrir las posiciones: [3], [4] y [5].
* 3 segundos = Como hay otro maíz en la posición [6], se pone un espantapájaros en la posición [7] que cubrirá las siguientes posiciones: [6], [7] y [8].
* 1 segundo = Los 1 que se encuentran entre 9, 6 y 3 son debido al tiempo en colocarlo, y el penúltimo 1 es debido a que hay otro maíz en la posición [9] y se debe poner ahí.

Comparando ambos tiempos, el algoritmo Greedy redujo el tiempo de ejecución en poco más del 60%, en este caso, la cadena utilizada fue de únicamente 10 elementos, pero si nos imaginamos un terreno de 1x100 o 1x1000, entonces el algoritmo Greedy ganaría por mucho. Para demostrarlo, se muestran en las figuras 9 y 10 los espantapájaros necesarios y los segundos totales en colocar los espantapájaros para un terreno que este lleno de cosechas para un terreno de 1x10 (comprobando si los cálculos realizados en la parte de arriba son correctos).

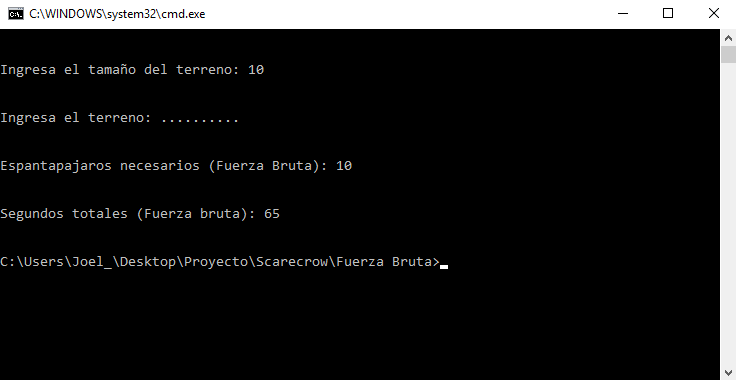


Figura . Espantapájaros y segundos totales (1x10 - Fuerza Bruta)



Figura . Espantapájaros y segundos totales (1x10 - Greedy)

A continuación, en las Figuras 11 y 12 se muestra la misma impresión de pantalla, pero para un terreno con una longitud de 1x50.

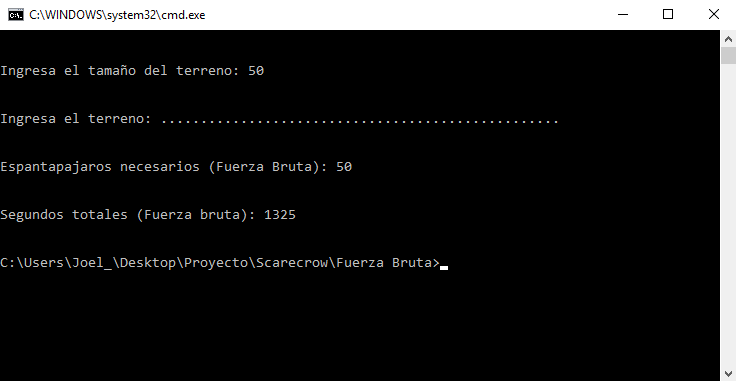


Figura . Espantapájaros y segundos totales (1x50 - Fuerza Bruta)



Figura . Espantapájaros y segundos totales (1x50 - Greedy)

Como podemos ver, el tiempo se disminuye bastante con un algoritmo Greedy, sin embargo, para ilustrar mejor esta gran diferencia, en la Figura 13 se observa una gráfica comparando ambos algoritmos en cuanto a segundos para ver la velocidad de resolver el problema de cada uno.

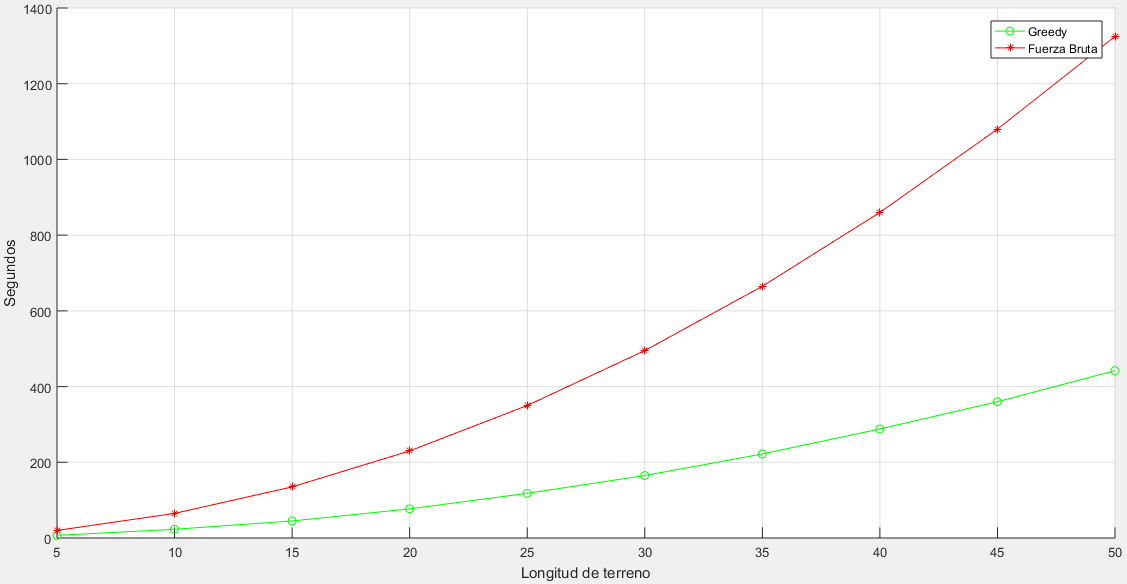


Figura . Comparación Greedy vs Fuerza Bruta

**Bear and Row 01 [3]**

* **Fuerza Bruta**

Para resolver el problema por fuerza bruta, habría que ir moviendo los soldados partiendo del soldado que se encuentre más a la izquierda, hacia lo más a la derecha que se pueda, y así sucesivamente hasta que todos lleguen a la parte derecha de la fila.

Como sabemos, si hay un soldado en la posición que sigue, el soldado que está atrás no puede seguir avanzando hasta que el otro se mueva, es decir, un soldado no puede saltar a otro en la fila.

El peor caso, sería que en una fila estuvieran intercalados 1 soldado y 1 obstáculo, ya que se tendría que mover muchas veces cada soldado (recordemos que darle la orden cuesta 1 segundo, y cada posición que se mueve cuesta otro segundo), así que, suponiendo que tuviéramos una fila de 10 soldados, comenzando con 1 soldado, 1 obstáculo y así sucesivamente, el tiempo total estaría dado por:

2 + 2 + 2 + 2 + 2 + 2 + 2 + 2 + 2 + 2 + 2 + 2 + 2 + 2 + 2 = 30

El total de segundos para el algoritmo por fuerza bruta es de 30 segundos, sin embargo, hay que analizar el tiempo que tardaría el mismo problema con un algoritmo Greedy para tener un punto de comparación.

* **Greedy**

En este caso, para la solución Greedy, lo que se debe hacer es exactamente lo contrario, es decir, ahora ir moviendo los soldados (leyendo la cadena) de derecha a izquierda, es decir, que el soldado que está hasta la derecha se va a mover el número de ceros que tenga enfrente + 1 (el tiempo en el que le dan la orden de moverse), por lo tanto, el soldado que sigue atrás de él (si no es un soldado que está inmediatamente detrás de él), se va a mover el número de ceros que tenga enfrente + 1 (el tiempo en el que le dan la orden) + el número de posiciones que se haya movido el soldado frente a él.

En caso de que 2 o más soldados estén formados exactamente uno detrás de otro, el soldado más a la derecha se va a mover el número de ceros que tenga enfrente + 1 (el tiempo en el que le dan la orden) y el/los que estén inmediatamente atrás de él, se van a mover las mismas posiciones que se movió el que estaba más a la derecha.

En una cadena de 10 soldados con los soldados intercalados, el número de segundos totales estaría dado por:

2 + 3 + 4 + 5 + 6 = 20 segundos

A continuación, en las Figuras 14 y 15, se puede observar el problema resuelto con fuerza bruta y con un algoritmo Greedy para la cadena propuesta (“1010101010”).

Como ya sabemos resolver el problema, debemos de contar los números 1 consecutivos (que representan a los soldados), y posteriormente contar los ceros que hay frente a los n números 1. Al final, debemos multiplicar el numero de 1 consecutivos que hubo, por el número de ceros que contamos + 1 (por el tiempo que se tarda en darle la orden al soldado).

Para el caso del algoritmo Greedy (para minimizar), debemos de hacer exactamente lo contrario (ya que haremos lo mismo pero leyendo la cadena de derecha a izquierda) pero al revés.



Figura . Fila de soldados, segundos totales (1x10 - Fuerza Bruta)



Figura . Fila de soldados, segundos totales (1x10 - Greedy)

A continuación, se muestran ambos algoritmos de la misma manera, para una cadena más larga (fila de soldados de 50).

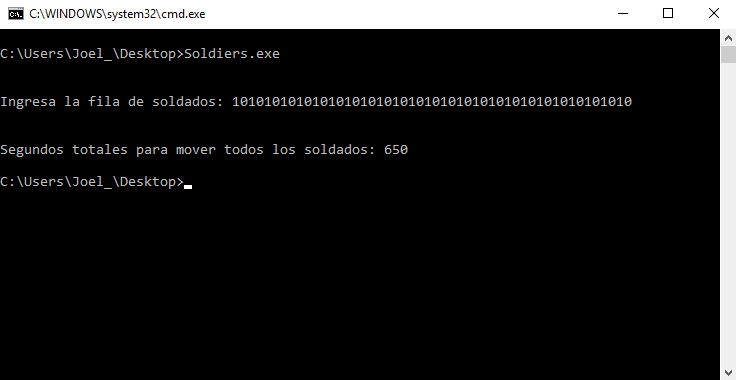


Figura . Fila de soldados, segundos totales (1x50 - Fuerza Bruta)



Figura . Fila de soldados, segundos totales (1x50 - Greedy)

Para poder apreciar mejor la diferencia de tiempos, en la Figura 18 se muestra una gráfica comparando Fuerza Bruta vs Greedy.

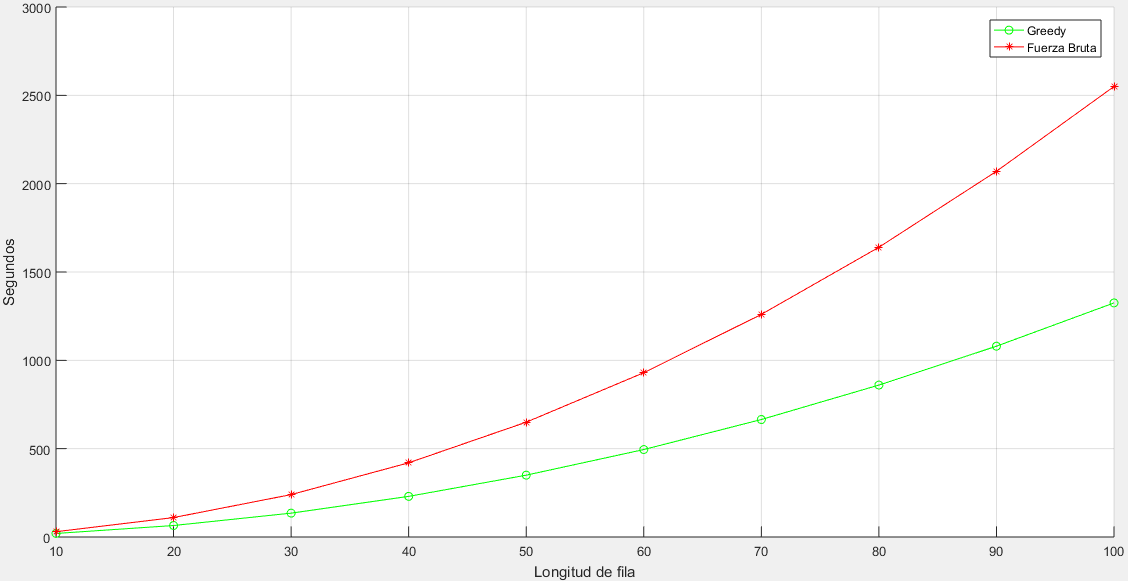


Figura . Comparación Greedy vs Fuerza Bruta

# Software (librerías, paquetes, herramientas):

* Visual Studio Enterprise 2017: IDE [4]
* Bunifu: .NET UI Framework [5]
* MATLAB R2016a [6]

# Procedimiento:

\* Diagramas de Flujo / Diagrama a Bloques  
\* Agregar detalles (paso a paso) del procedimiento de manera que cualquier persona que lea pueda repetir el experimento.

# Resultados

En las siguientes figuras, se observa una cadena de ejemplo, cabe resaltar, que está predefinido que si no se llenan todas las casillas disponibles, el resto se rellena con tierra infértil.



Figura . Cadena de ejemplo para problema de Scarecrow.

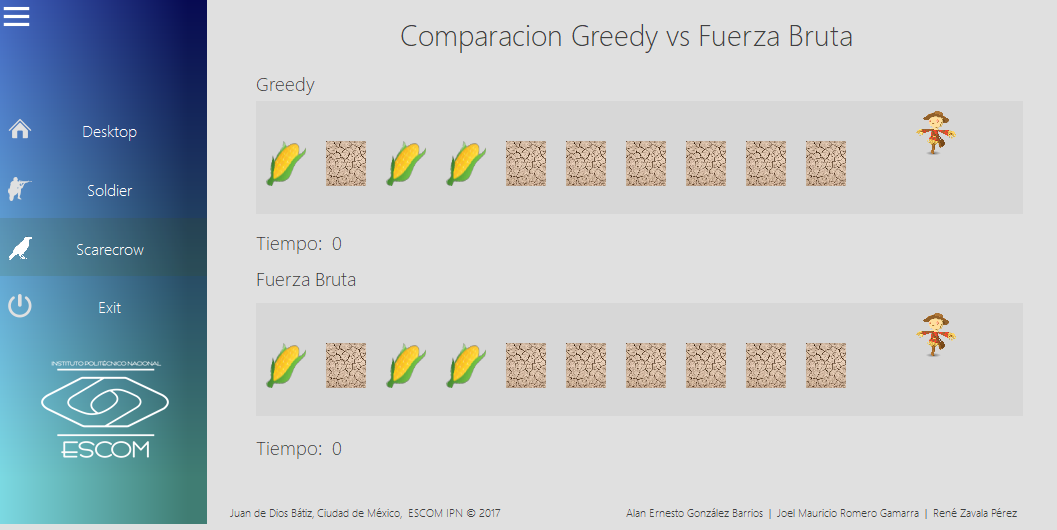


Figura . Cadena completa para animar

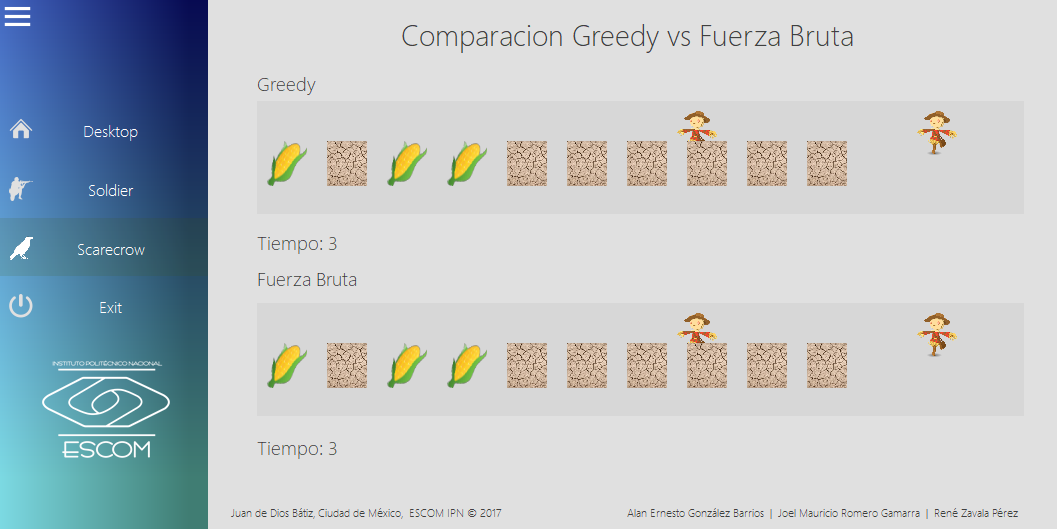


Figura . Animación (1)

Como se puede observar, al momento de la captura, ambos dicen 3 segundos, debido a que cada movimiento se toma como 1 segundo, es decir, los tiempos que se contarán es cada desplazamiento a la izquierda y 1 extra en lo que se coloca el espantapájaros (la colocación se simula con un desplazamiento hacia abajo).

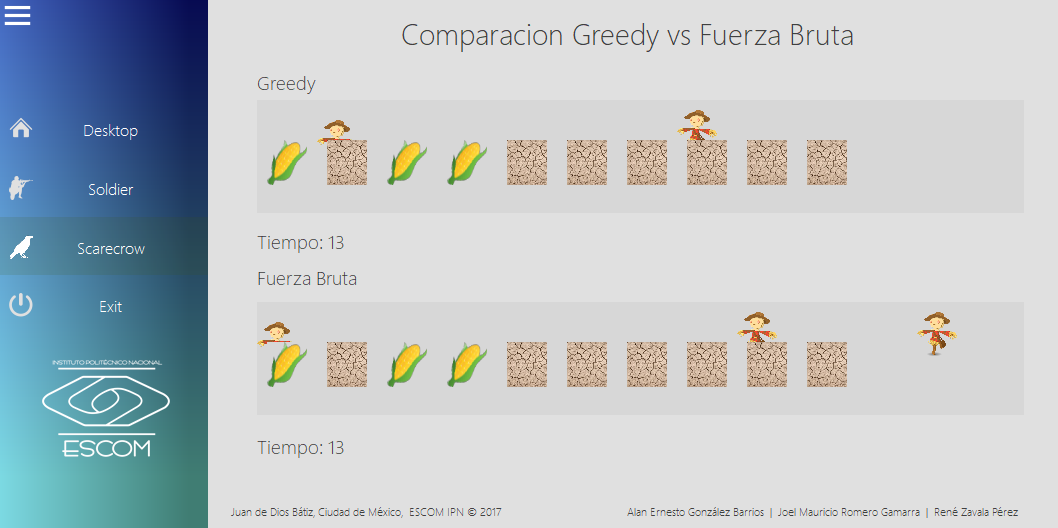


Figura . Animación (2)

Ahora, podemos ver que el tiempo es el mismo, sin embargo, en el espacio para Greedy uno de los espantapájaros está 1 posición más adelante que el de fuerza bruta y no van a la par como al principio, ya que el espantapájaros de fuerza bruta se colocó 1 posición más adelante y esto provoca un retardo en el tiempo.

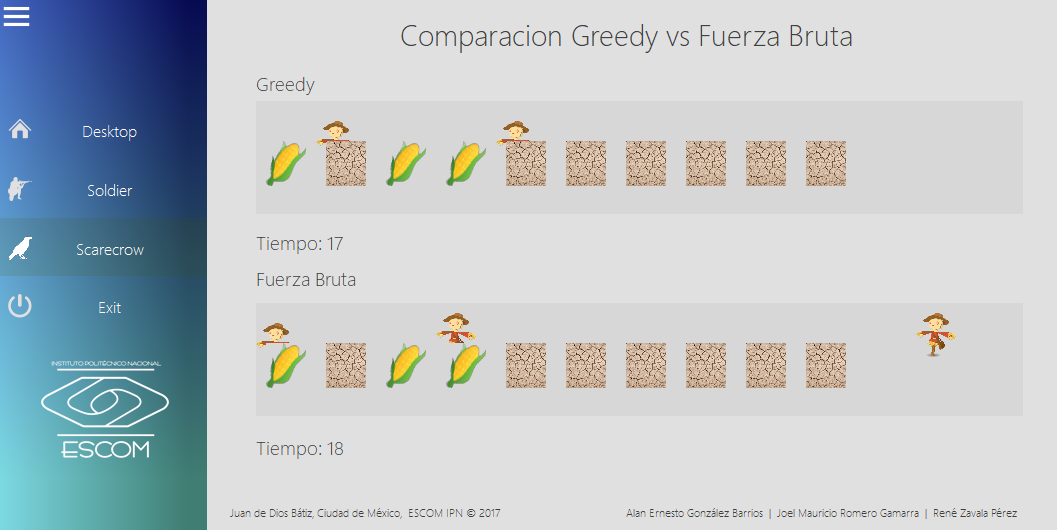


Figura . Animación (3)

En este momento, el algoritmo Greedy ya terminó, mientras que el algoritmo de Fuerza Bruta sigue animando, además, se puede observar en la parte derecha que aún falta 1 espantapájaros por ser colocado, en la siguiente Figura, se muestra el tiempo final de ambos algoritmos junto con la posición final de los espantapájaros.

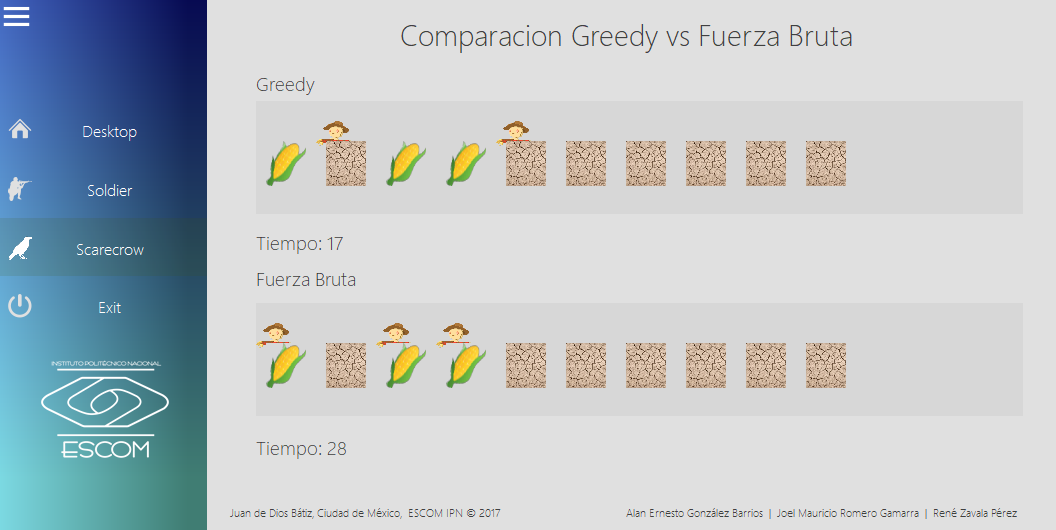


Figura . Animación (4)

Como se puede ver, los tiempos son muy distintos, y además en Fuerza Bruta se obtuvo la solución más tonta que cubra el terreno y además sea muy tardada.

Ya que se animó una cadena de ejemplo, a continuación, se muestran otros 3 ejemplos de cadenas distintas mostrando el tiempo de ejecución de cada algoritmo, pero únicamente la pantalla final que muestra los tiempos de cada algoritmo y la posición de los espantapájaros.

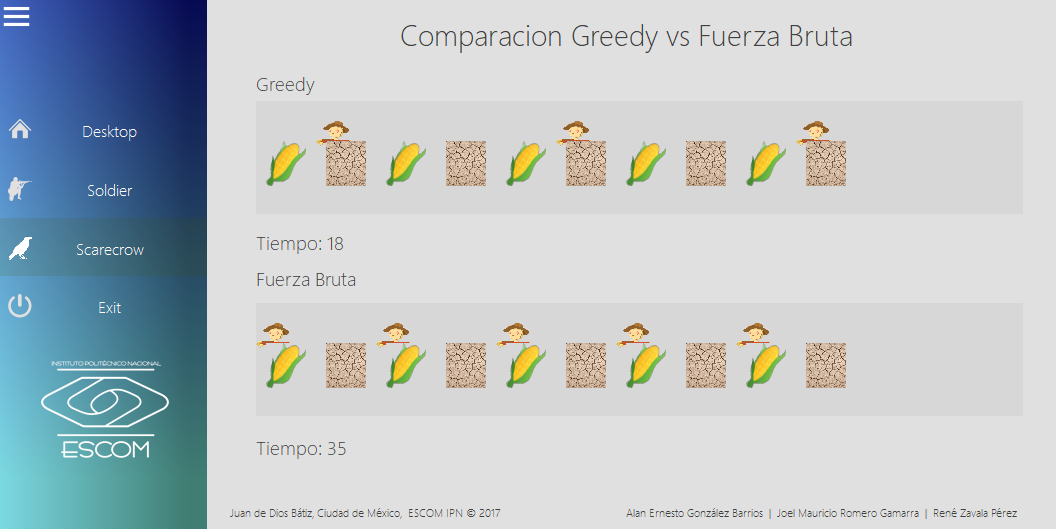


Figura . Ejemplo 2

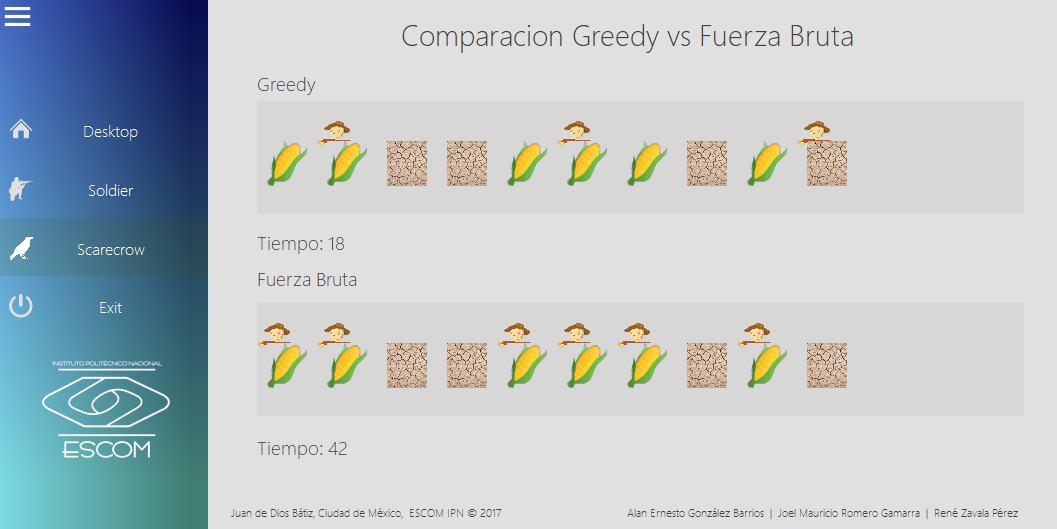


Figura . Ejemplo 3

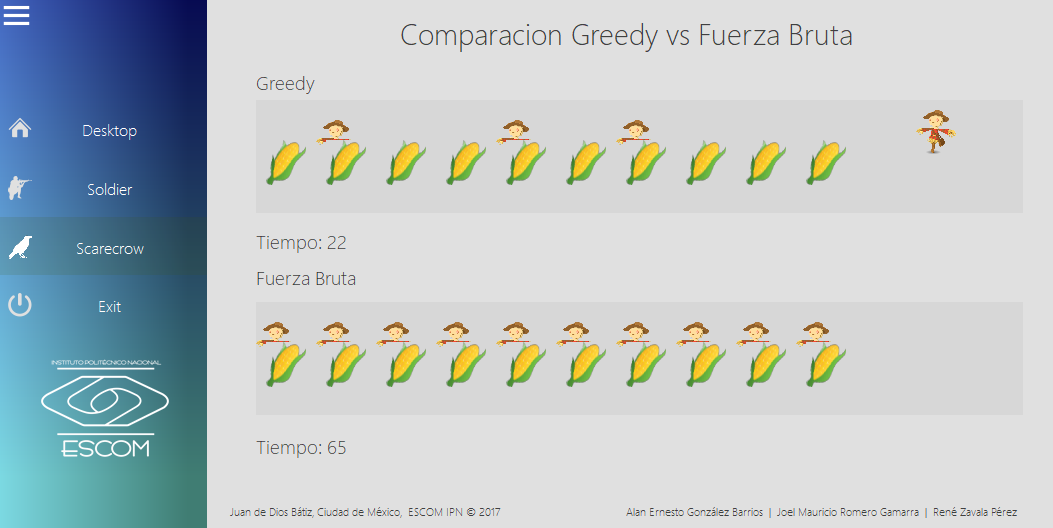


Figura . Ejemplo 4

# Conclusiones:

**René Zavala Pérez**

Aquí escribe tus conclusiones

**Joel Mauricio Romero Gamarra**

El uso de algoritmos Greedy es muy importante debido a que el tiempo de cómputo es muy corto, ya que resuelven el problema en 1 solo ciclo y eso hace que la ejecución del mismo sea muy rápida, todo esto tomando la mejor decisión en el momento con la poca información que se tenga en ese momento y así formar la mejor solución para el problema completo.

Al ser problemas de optimización, tienen muchas aplicaciones para problemas de la vida real, ya que día con día tenemos que tomar estas decisiones, por ejemplo, en algo tan simple como llegar a algún destino, sea la escuela, el trabajo, una cita, etc. Siempre buscamos minimizar el costo y/o el tiempo para llegar, o cuando queremos comprar algo, muchas veces nos pasa que hay promociones en el producto que estamos buscando y debemos saber si nos conviene o si no.

Creo que es importante tener bien claro el concepto, ya que, muchas veces estamos acostumbrados a resolver los problemas que se nos presentan (hablando en cuanto a las prácticas) como se nos ocurre primero sin pensar si puede haber una mejor solución, así que atacamos al problema como una solución por fuerza bruta cuando en realidad tiene una respuesta más simple que no pudimos ver y esto provoca que nuestros programas tengan un tiempo de ejecución muy alto.

# Referencias:

**[1]** Edgardo Adrián Franco Martínez, “Análisis de Algoritmos: Algoritmos Ávidos”, Noviembre 2017. [Online]. Disponible en: [http://www.eafranco.com/docencia/analisisdealgoritmos/files/09 /Tema09.pdf](http://www.eafranco.com/docencia/analisisdealgoritmos/files/09%20/Tema09.pdf)

**[2]** UVa Online Judge, “12405 - Scarecrow”. [Online]. Disponible en: <https://uva.onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=657&page=show_problem&problem=3836>

**[3]** Kamil Debowski, “Bear and Row 01”, April 2017. [Online]. Disponible en: <https://www.codechef.com/problems/ROWSOLD>

**[4]** Microsoft, “Visual Studio Enterprise”. Disponible en: <https://www.visualstudio.com/es/vs/enterprise/>

**[5]** Bunifu, “.NET UI Framework – Empowering Developers to build faster, beautiful with less”. Disponible en: <https://devtools.bunifu.co.ke/>

**[6]** Math Works, “MATLAB”. Disponible en: <https://es.mathworks.com/products/matlab>.

# Código

Incluir todo el código fuente, comentar todo el código reutilizado y mostrar referencias.

Se debe usar el siguiente link para darle formato al código.

<https://tohtml.com/c/>