Proyecto 2:
Implementación de la Optimización por enjambre de particulas para el Problema de la galería de arte

Carlos Gerardo Acosta Hernández

Heurísticas de optimización combinatoria 2017-2
Facultad de Ciencias UNAM

Como segundo proyecto del seminario, decidí implementar el problema de la galería de arte[1] en su versión con guardias para los vértices de un polígono simple, junto con la heurística de optimización por enjambre de partículas[2] en su versión binaria para la elección de dichos guardias. En las siguientes secciones abordaré algunos conceptos importantes para la implementación que realicé, tanto de la heurística, como del problema de mi elección; asimismo, presentaré algunos detalles sobre los componentes de mi sistema, las entradas y la experimentación a la que lo he sometido hasta el momento. Por último, al final del documento proporciono las instrucciones y herramientas necesarias para construir el sistema y manejarlo con éxito.

1. Preliminares

1.1. Problema de la galería de arte

El *Problema de la galería de arte* -también conocido como problema del museo-, se puede fácilmente explicar como un problema común en la vida real: Para la disposición de guardias de seguridad en una galería de arte, encontrar una configuración en la que toda el área de la galería esté resguardada por un número mínimo de guardias. Este problema vio la luz en el año 1973, cuando el matemático estadounidense Victor Klee, inquirió a la comunidad científica sobre cuántos guardias han de ser necesarios para vigilar un polígono de n vértices. Por el momento diremos que un polígono está vigilado si para todo $p \in P$, el segmento de línea formado por $g \in G$ y p se mantiene *dentro* del polígono, donde P es el conjunto de los puntos en el plano dentro del polígono y G un conjunto de puntos en el plano. En respuesta a la interrogante, Václav Chvátal, propuso propuso y demostró un teorema para este problema en el que se establecía una cota superior de $\lfloor \frac{n}{3} \rfloor$ del mínimo número de guardias **necesarios** para vigilar un polígono de n vértices. Es decir que no siempre es necesario ese número de guardias pero siempre es suficiente. Sin embargo, esta cota

[1] Art Gallery Problem
[2] Particle Swarm Optimization

sugiere que las soluciones factibles que permanecen cercanas a la suficiencia, siguen estando muy alejadas de una configuración óptima para los guardias. El problema de hallar tal configuración pertenece a la clase de complejidad *NP-difícil*, incluso para polígonos simples.

Con el fin de definir el problema considerado lo más generalmente posible, es conveniente revisar lo siguiente: Sea P un polígono con n vertices. Sea h el número de "hoyos" en P, para este proyecto he decidido trabajar para $h = 0$, es decir que se trabaja con polígonos simples, ya sean convexos o cóncavos. Para cualesquiera dos puntos $p, q \in V(P)$, el conjunto $V(P)$ de los vértices del póligono, se dice que son "visibles" entre ellos si el segmento de recta pq se encuentra "dentro" del polígono. Para definir esta contención en el polígono he considerado estas dos condiciones:

- Que pq no intersecta ninguna de las aristas del polígono y,
- el punto medio de pq se encuentra dentro del polígono.

Ahora bien, denotemos el conjunto de visibilidad de un punto $p \in V(P)$ como $M(p) = \{q \in V(P) \mid q \text{ es visible a } p\}$, de manera que para un conjunto de puntos $G \subseteq P$, $M(G)$ estará definido como $M(G) = \cup_{r \in G} M(r)$. Determinaremos que un conjunto $G \subset V(P)$, vigila a todo el polígono si $M(G) = V(P)$. El objetivo entonces es encontrar un conjunto G vigilante de P, tal que su cardinalidad sea mínima, aquí nos centraremos cuando menos en hallar una cardinalidad que esté por debajo de la cota establecida, de ser posible.

1.2. Optimización por enjambre de partículas

La optimización por enjambre de partículas es una heurística inspirada en el comportamiento social colectivo de agrupaciones o aglomeraciones de seres vivos -manadas, colonias, enjambres, entre otros. Utilizando una población de individuos, el algoritmo tiene por objetivo que cada individuo represente una solución al problema mediante su posición, codificada en un vector multidimensional del espacio de búsqueda. Además, el individuo debe tener cierta velocidad que le permita actualizar su posición, codificada similarmente en otro vector. La idea general de la búsqueda es que la actualización de la posición de una partícula no sea un mero factor aleatorio, sino que incluya cierta influencia de la mejor solución de toda la población, digamos un individuo destacado que "lidere" al colectivo, y la mejor posición que haya encontrado en iteraciones anteriores que no necesariamente es la misma que mantiene en la más reciente. Por tanto, la *exploración* es regida por el factor de aleatoriedad y la *explotación* mediante las posiciones que influyen en la actualización mencionada. De tal suerte que cada partícula tenga posibilidad de explorar con cierta libertad el espacio de búsqueda pero se mantenga cerca del enjambre, que avanza en conjunto hacia valores óptimos.

La heurística, propuesta por Russel Eberhart y James Kennedy en 1995, fue originalmente concebida para espacios de búsqueda contínuos, sin embargo, se han propuesto versiones discretas del algoritmo desde entonces, entre ellas, su versión binaria que implementé para este proyecto. En esta adecuación, cada entrada en el vector de posición de una partícula es un *bit* y una entrada en el vector de velocidad representa la probabilidad de que la entrada correspondiente en el vector de posición se establezca como 1 o 0.

El algoritmo siguiente describe el funcionamiento general de la heurística, por lo que sólo será necesario hacer un par de anotaciones posteriores.

ALGORITHM 1: Particle Swarm Optimization (PSO)

Input: n-vertices polygon
Output: Best particle in swarm
$t \leftarrow 0$; $E^t \leftarrow randomValidPopulation(s)$;
repeat
  $t \leftarrow t + 1$;
  for *each* p_i^t *in* E^t **do**
    for *each* $d = 0, 1, ..., n-1$ **do**
      $r_p \leftarrow U(0,1)$, $r_g \leftarrow U(0,1)$
      $v_{i,d}^{t+1} \leftarrow \omega v_{i,d}^t + \varphi_p r_p (b_{i,d}^t - p_{i,d}^t) + \varphi_g r_g (E_{g_d}^t - p_{i,d}^t)$
    end
    update particle's position: $p_i^{t+1} \leftarrow p_i^t + v_i^t$
    if $f(p_i^t) < f(b_i^t)$ **then**
      $b_i^{t+1} \leftarrow p_i^t$
    if $f(p_i^t) < f(E_g^t)$ **then**
      $E_g^{t+1} \leftarrow p_i^t$
  end
until $t = maxGen$;
return E_g

Aquí vale la pena señalar que U es un generador de números pseudoaleatorios entre [0,1] $\in \mathbb{R}$, b_i es la mejor posición de la i-ésima partícula del enjambre, ω es el peso o inercia que se ejerce sobre la partícula que determina en buena medida su "movimiento", φ_p y φ_g son dos constantes positivas. Estas tres últimas variables, junto con el tamaño s del enjambre o población y el máximo de iteraciones $maxGen$ establecido -que en este caso funge como condición de terminación, pero puede ser sustituido por otro que se considere más conveniente-, son parte de los parámetros que afectan el comportamiento de la heurística y que deben ser explorados durante la experimentaación para mejores (y reproducibles) resultados.

Como ya había mencionado, deben hacerse un par de cambios a la versión original, considerando que por ejemplo, no podemos sumar el vector de posición $p_i \in \{0,1\}^n$ con el vector de velocidad $v_i \in \mathbb{R}^n$ directamente. Es por ello, que tanto la actualización de la velocidad como de la posición de la partícula se ven definidos por estas nuevas ecuaciones:

$$v_{i,d}^{t+1} = \frac{1}{1 + e^{-v_{i,d}^t}} \tag{1}$$

$$p_{i,d}^{t+1} = \begin{cases} 0 & \text{if } r_{i,d} < v_{i,d}^t \\ 1 & \text{en otro caso} \end{cases} \tag{2}$$

Donde $r_{i,j} \in \mathbb{R}$ es un número pseudo-aleatorio entre [0,1].

2. Especificación

En la presente sección presentaré las herramientas utilizadas para la construcción del sistema, el diseño utilizado para el código junto con la estrategía general utilizada para la implementación y finalmente algunas de las instancias que se calcularon como entrada de la heurística.

2.1. Herramientas

El sistema consta de los siguientes componentes:

- **Lenguaje de programación:** Scala 2.12.1
- **Sistema de construcción:** SBT (*Scala Build Tool*) 0.13.13
- **Documentación:** ScalaDoc 2.12.1
- **Pruebas unitarias:** ScalaTest 3.0.1
- **Control de versiones:** Para mantener el control de versiones se utilizó Git 2.11.0 y el repositorio en línea se encuentra alojado en GitHub.

Estructura

El proyecto está organizado con la jerarquía de directorios que se especifica para proyectos de *SBT* (el sistema de construcción). El árbol desde el directorio raíz, debe verse como:

```
PSO-AGP/
├── build.sbt
├── doc/
├── instances/
├── lib/
├── lib_managed/
├── project/
├── README.md
└── src/
```

En la carpeta **doc**/, se encuentra este documento que estás leyendo, y su código fuente en LaTeX.

En la carpeta de **lib**/, se incluyen las bibliotecas externas al sistma de construcción y al lenguaje de programación de las que depende el proyecto. Para este proyecto en particular tenía contemplado utilizar *JavaFX* o en su defecto *ScalaFX* para una interfaz gráfica interactiva, sin embargo aún no hay implementación lista para tal propósito.

En el directorio **instances**/, se encuentran las instancias de entrada del sistema. Archivos con terminación *.pol*, en referencia a la palabra polígono y un subdirectorio **solutions**/, con imágenes en formato *SVG* producidas al termino de cada ejecución, que deben poseer el mismo nombre que

la instancia de la que provienen. Presentaré el formato de las instancias en la última sección que corresponde a la experimentación con dichos archivos (Sección 2.4).

Todo el código fuente referido a la implementación de la heurística se encuentr en el directorio **src**/. Desde ahí es posible exploar el código. Es importante recalcar que se incluye una carpeta para código escrito en *Java*, sin embargo, se encontrará vacía, pues la implementación está realizada por completo con la sintáxis de *Scala*. Hay dos subdirectorios en dicha sección, pero lo retomaremos más adelante en diseño (Sección 2.2).

En **project** están definidas las dependencias del proyecto en un archivo de nombre **Dependencies.scala**, donde se definen las características del sistema de construcción (puede señalarse, por ejemplo, una versión específica de *SBT* que será descargada en cuanto se inicialice el proyecto por la versión actual que tenga el sistema). Asimismo pueden agregarse todas las dependencias que sean necesarias, por el momento sólo se define la de *ScalaTest* para pruebas unitarias.

2.2. Diseño

Como se mencionaba en la sección anterior, la implementación está dividida en dos paquetes principales, subdirectorios del paquete raíz de código fuente de *Scala*.

```
src/main/scala/
├── AGP/
├── BPSO/
├── Controlador.scala
├── Main.scala
└── SVG.scala
```

El paquete **AGP** contiene todo los referente al modelado de polígonos, mientras que **BPSO**[3] contiene todas las implementaciones necesarias para la heurística. A diferencia del proyecto 1, he decidido no hacer *traits* previo al modelado, pues resultaban altamente dependientes del problema, por lo que una modularización para cada componente de la heurística me pareció suficiente. Aunque claro, conserva cierto parecido con la diligencia de ese primer proyecto.

1. CTerminacion.scala
2. FuncionDeCosto.scala
3. GeneradorVerificador.scala
4. Enjambre.scala
5. BPSO.scala
6. Particula.scala
7. Poligono.scala
8. Punto.scala

Al respecto de los archivos libres en la carpeta, como su nombre indica *Controlador.scala* es el código encargado de iniciar la ejecución de la heurística con la entrada provista, *Main.scala* es el

[3]Se refiere a *Binary Particle Swarm Optimization*

código encargado de la interfaz de usuario para el uso del sistema y *SVG.scala* una herramienta de dibujo para las instancias (los polígonos). Para información más detallada de la implementación, favor de revisar *target/scala-2.12/api/***index.html** con la *API* generada por *ScalaDoc*.

2.3. Implementación

La implementación de la heurística está construida como se explica en 1.2, donde el vector de posición binario de una partícula refiere a una configuración de guardias en los vértices del polígono. Si el vector en la posición i esta asignado como verdadero, entonces hay un guardia en el i-ésimo vértice del polígono.

Función de costo

La función de costo es un trabajo en proceso, puesto que aún es necesario analizar la experimentación y no he podido encontrar un trabajo previo específico para este problema que signifique una base teórica fuerte para sustentar su idoneidad. Sin embargo, ha probado su utilidad para las instancias propuestas, permitiendo encontrar resultados factibles al menos menores a la cota superior del número de guardias. Para presentarla, es preciso señalar que en la implementación, todo objeto instancia de la clase partícula, posee, entre otros atributos fundamentales, un arreglo de valores *double*, de la misma longitud que su vector de posición. Para cada entrada i del arreglo, refiere al número de guardias para los que es visible el i-ésimo vértice del polígono en esa configuración de guardias propuesta por esa partícula. Dado que no consideré conveniente calificar a las partículas directamente por la cantidad de guardias que poseen porque eventualmente llegaría a soluciones en definitiva no factibles, la oportunidad de minimización se realiza considerando ese arreglo de visiblidad, como principal factor.

Sea $s \in E$, una partícula del enjambre, $nG(s)$, el número de guardias en la posición de la partícula y M su arreglo de visibilidad, la función de costo estará definida como:

$$f(s) = \frac{nG(s)}{n} \cdot \sum_{i=0}^{n-1} M_s(i) \tag{3}$$

Es claro que el primer factor entre menor número de guardias, permitirá que la evaluación sea menor, sin embargo, si tenemos por ejemplo una configuración con 0 guardias, la función de costo evaluaría a 0, con lo que se complica el objetivo. Es por ello que este arreglo de visibilidad debe tener en todas sus entradas, previamente a la ejecución de las iteraciones, valores superiores al máximo valor de vigilantes por punto, que por el momento está implementado como $M(i) = (n+1) * 99$ al momento de su inicialización. No es hasta que el i-ésimo punto es visible para algún guardia que $M(i)$ toma valor 1, e irá incrementando conforme aparezcan nuevos guardias en la configuración que puedan vigilarlo, o bien decrementando en caso de que cambie la configuración y pierda vigilantes. En caso de que llegue a 0 en ese decremento, se "reinicializará" esa posición en el arreglo de visibilidad al valor propuesto de inicio. Podemos ver entonces que $1 \geq M(i) \leq n$ o $M(i) = (n+1) * 99$.

Se espera que con esta función de costo, se preserve la relación de orden entre las partículas y sea posible decidir un camino adecuado para continuar la búsqueda.

2.4. Experimentacion

Archivo de configuración

Dentro de la carpeta de **instances**/ he incluído algunas de las instancias que probé para el sistema, ya con el formato requerido para la interfaz de usuario. Todo archivo de entrada debe contar en su primera línea con 6 argumentos separados por comas y sin espacios, en el siguiente orden

1. La semilla para el generador de números pseudoaleatorios
2. El tamaño del enjambre, i.e., el número de partículas de la población
3. El máximo número de iteraciones
4. ω, el peso o la inercia constante que se ejerce sobre las partículas
5. φ_p, el factor de influencia de la mejor posición de la partícula
6. φ_g, el factor de influencia de la posición de la mejor partícula en el enjambre

Es decir que la primera línea debe tener la siguiente sintáxis:

```
[semilla],[tamaño_enjambre],[max_iter],[omega],[phi],[phi2]
```

Las siguientes líneas deben corresponder a un vértice del polígono, siendo cada i-ésimo punto adyacente con el (i+1)-ésimo, por supuesto módulo n, es decir que el primero es adyacente con el último. Cada vértice debe constar de una coordenada x y una coordenada y en $(x, y) \in \mathbb{R}^2$, separadas por una coma y sin espacios, sin parentesis ni nada más que un salto de línea luego de la y.

```
x,y
x2,y2
x3,y3
...
```

A continuación se encuentran algunos ejemplos de las instancias más relevantes con los parámetros para hallar la mejor solución hasta el momento del sistema.

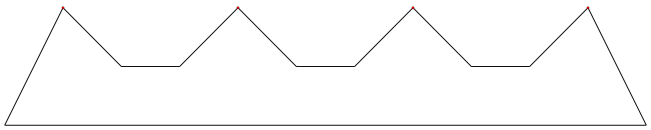
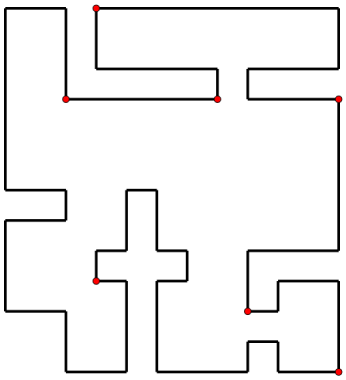


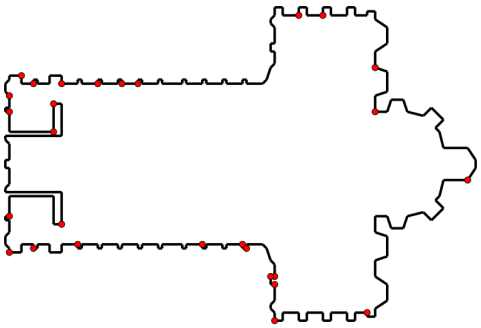
Figura 1: Solución factible a instancia *peine.pol*.

1. Semilla = 1
2. Tamaño del enjambre = 100
3. Máximo de iteraciones = 100
4. $\omega =$, $\varphi_p =$,$\varphi_g = 0.1, 0.1, 0.1$



1. Semilla = 23
2. Tamaño del enjambre = 2000
3. Máximo de iteraciones = 100
4. ω = -1.5, φ_p = 0.0003,φ_g = 0.004

Figura 2: Solución factible a instancia *galeria.pol*.



1. `Semilla = 1`
2. `Tamaño del enjambre = 10000`
3. `Máximo de iteraciones = 100`
4. ω `= -1.2,` φ_p `= 0.0001 ,`φ_g `= 0.01`

Figura 3: Solución factible a instancia *StSerninH.pol*.

3. Manejo del sistema

Descarga del proyecto

El proyecto se encuetntra alojado en línea en *GitHub*, para descargarlo basta ejecutar *Git* en cualquier directorio del sistema de archivos.

```
$ git clone https://github.com/Pernath/PSO-AGP.git
```

Se creará un directorio con la estructura definida en la sección 2.2.

Compilación y ejecución

Para las operaciones siguientes se supone que el sistema operativo ya cuenta con el sistema de construcción *sbt* instalado.

En primer lugar llamamos al sistema de construcción, que buscará el archivo **build.sbt** y establecerá la ruta actual como la ruta del proyecto.

```
[user@host PSO-AGP]$ sbt
```

Esto nos llevará a un *prompt* desde el que podremos llamar los siguientes comandos de acuerdo a lo que necesitemos.

Para compilar el proyecto y verificar que todo está listo.

```
> compile
```

Posteriormente, será posible ejecutar el programa con el archivo de configuración como argumento.

```
> run conf.txt
```

Este archivo debe cumplir el formato especificado en 2.4.

Empaquetado

Una manera de empacar la implementación es ejecutar el siguiente comando:

```
> package
```

Sin embargo, este no incluirá las dependencias externas en el *JAR* generado bajo la ruta *target/scala-2.12/***AGP.jar**, por lo que no será portátil y dependerá siempre de la estructura de directorios del sistema.

Por fortuna, también incluí la opción para generar un *stand-alone JAR* que nos permitirá mover líbremente el ejecutable de *Java* y ejecutarlo desde cualquier directorio. Esto se logra con el siguiente comando y podrá encontrarse el ejecutable en la misma dirección mencionada anteriormente:

```
> assembly
```

Este también ejecutará las pruebas unitarias incluídas en **src/test/**.

JAR

Para ejecutar el resultado del ensamblaje, debemos ubicar el archivo *JAR* en la carpeta mencionada.

```
[user@host scala-1.12]$ java -jar AUTSP.jar conf.txt
```

Es importante recordar que este paquete será bastante más rápido en ejecución que realizando un *run* sobre el "prompt" de *sbt*.

Referencias

[1] Arora, S., Barak, B. *Computational complexity: a Modern Approach.* Beijing: World Publishing Corporation. 2012.

[2] Reza, N., Mohades, A., *Solving minimum-vertex guard in art-gallery problem by a new heuristic algorithm* http://scholarly-journals.com/, Department of Computer Science and Mathematics, Amirkabir University of Technology, Tehran, Iran. 2013.

[3] Amit, Y., Mitchell, J., Packer, E., *Locating guards for visibility coverage of polygons.* http://researcher.ibm.com/, International Journal of Computational Geometry & Applications, World Scientific Publishing Company, 2012.

[4] Ahmadieh, G., *A Novel Binary Particle Swarm Optimization* . http://zeus.inf.ucv.cl/, Proceedings of the 15th Mediterranean Conference on Control & Automaton, Athens, Greece. 2007.