Algoritmo de optimización de busqueda de pinguinos para problema el k-MST

Vidal Aguilar Diego Jesus

Universidad Nacional Autonoma de México
Facultad de Ciencias

Abstract

El presente trabajo es una implementación de busqueda local del algoritmo de optimización de busqueda de pinguinos para la solución del problema k-MST. La implementación corresponde a la discretización del algoritmo de optimización de busqueda de pinguinos [1]. Manipulando la discretización para que realice una busqueda local y su respectiva implementación en Rust.

Introducción

El trabajo actual consiste en la implementación de la optimización de busqueda de pinguinos para aproximar una solución del problema de k-MST de manera discreta con busqueda local, al igual que los resultados encontrados a través de la experimentación con la variación de los parametros como lo son el numero de pinguinos, la cantidad de grupos, la cantidad de clavados y los niveles que realiza la heuristica. Con esto se espera poder aproximar a la mejor solución para una instancia del problema seleccionada o en el mejor de los casos a cualquier instancia del problema.

Problema a Resolver

El problema de k-MST conocido también como arbol generador minimo de tamaño k, donde k es el número de vertices del arbol. El árbol generador minimo de k vertices es un problema el cual dada una grafica $G = (V,E)$ con un coste $c_e > 0$ para cada arista en E y un vertice raiz $r \in V$. Buscamos encontrar un arbol que conecta al menos k vertices a la raiz mientras minimiza el costo total de las aristas del arbol.

Desarrollo

Para este problema se hizo uso de la heurística optimización de busqueda de pinguinos, la optimización de pinguinos es una técnica probabilista propuesta en 2013, la cual simula el proceso de caza de los pinguinos. Buscando emular este procedimiento de caza para lograr la optimización de soluciones de problemas de algoritmos.

En particular está heuristica simula el proceso de tal forma que empatiza con *PSO* a través de cada uno de los pinguinos busca aproximar cada una de las soluciones que mantiene cada uno de los pinguinos hacia las mejores soluciones, tal cual el comportamiento de las heuristicas basadas en *PSO*.

Esto nos lleva a comprender el proceso como lo siguiente, la heuristica consiste en emular el proceso de caza de los pinguinos, proceso el cual se basa en la busqueda por grupos, dado un conjunto de pinguinos que forman una población, se dividirán en grupos que buscarán cazar en distintos lugares del espacio.

Una vez realizamos esta división de los pinguinos en grupos, cada uno de estos pinguinos se sumerjera al agua y buceará hasta cierto nivel de profundidad, una vez el pinguino se encuentra en ese nivel es que busca comida, si encuentra una gran cantidad de comida avisa a los otros pinguinos de su gruopo sobre la posibilidad de acercarse a esta posición para consumir un mayor

numero de pescados.
Los pescados en un nivel eventualmente se terminarán, por lo que en ese momento el pinguino procederá a bajar al siguiente nivel y todos los demás pinguinos bajaran respectivamente de nivel, esto se realizará hasta que ya no queden niveles de profundidad por los cuales bajar.
Es bajo este principio que trabaja la heuristica, debido a que el mayor peso de la heuristica radica en este proceso propuesto, podemos darnos cuenta que al estar basada en *PSO* el espacio de busqueda de la heuristica es un espacio continuo. Es este espacio continuo el que nos lleva a pensar en *PSO* como una heuristica que trabaja sobre problemas continuos.
No es algo que funcionará abiertamente con nuestro problema, debido a que este como otros problemas en computación son problemas en espacios discretos, por lo cual se necesito una forma de discretizar este tipo de problemas, por ello es que recurrí a otro enfoque desde otra heuristica la cuál trata de adaptar este tipo de heuristicas a una manera de discretizar los problemas.
La idea propuesta radica en generar de manera aleatoria probabilidades de formar parte del conjunto de vertices que forman parte del arbol de k vertices, es esta estrategía la que esta basada en tomar un arreglo de tamaño n, donde cada uno de los elementos corresponda a un vertice de la gráfica completa. Una vez que se ha tomado este arreglo, de manera aleatoria se asignará un valor de 0 a 1, donde los k menores valores formarán parte del conjunto de vertices del arbol.
Mientras que el resto de vertices son deshechados, esto nos permite formar un arbol de tamaño k, evaluarlo bajo la función de costo (función la cuál será dada posteriormente) y finalmente a través de una función matematica, aproximar las probabilidades dentro de cada segmento a las probabilidades de la mejor solución, lo cuál corresponde al movimiento de los pinguinos.
Si bien esta solución puede resultar interesante, el mayor problema radica en la forma en la que son actualizdas las posiciones de los pinguinos, está función de aproximación puede generar cambios tales que el movimiento de los pinguinos sea mayor a un solo vertice. Lo cual provoca que perdamos la busqueda local.
Debido a que necesitamos de esta busqueda local para la realización del proyecto es que se opto por dejar esta idea de lado y se procedió a proponer una nueva idea.
Podemos evidenciar facilmente que el proceso de la heuristica consiste en 3 pasos principales:

- **Paso 1:** Generar distintas propuestas de solución para el problema de manera aleatoria.
- **Paso 2:** Agrupar cada una de estas soluciones en grupos los cuales se acercarán eventualmente a la mejor solución del grupo.
- **Paso 3:** Los grupos se acercarán eventualmente al grupo con la mejor solución.

Es este procedimiento el que se busco emular con la discretización de la heurística asegurando la búsqueda local en este procedimiento. Para realizar esto debemos asegurar que con cada uno de los cambios que se realicen en las soluciones unicamente se cambie un vertice, asegurando de esta manera que la solución alcanzada después de realizar este intercambio de vertices sea un vecino de la solución original.
Es esto lo que nos indica que la solución debe de cambiar en un vértice durante cada ejecución, esto nos asegura la búsqueda y la generación aleatoria de las soluciones. Pero al momento de agrupar las soluciones, no hay manera de asegurar que al realizar este procedimiento las soluciones eventualmente se acerquen a la mejor solución del grupo.
Es por ello que al realizar este intercambio en los vértices que forman parte del arbol se añadirá un breve sesgo, sesgo el cuál consiste en la condición de agregar uno de los vértices que se en-

cuentren en la mejor solucion del grupo, esta proporsión comenzo con un 60 % de probabilidad de recibir un vertice de la mejor solución y un 40 % de recibir un vertice aleatorio.
A partir de esto es que aseguramos que los grupos se acercarán eventualmente a la mejor solución del grupo, pero seguimos sin cumplir el paso 3, los grupos de soluciones no se aproximan al grupo con la mejor solución.
Para solucionar esto es que nuestro algoritmo procederá a añadir un nuevo sesgo en las soluciones que nos permitirán simular este proceso, este proceso se realizará a través de un intercambio de soluciones o de pinguinos.
Sabemos que el pinguino con la mejor solucion en cada grupo es el encargado de guiar a los pinguinos a mejores soluciines (más pescados). Este tipo de implementación es lo que nos permite que al peor grupo con el peor pinguino le agregaremos un nuevo sesgo, el peor pinguino será intercambiado por el mejor pinguino de todos los grupos.
Es decir, el mejor pinguino de todos compartirá la posición de sus pescados con el pinguino que tenga la peor posición de pescados. Esto nos asegura que la peor solución se convertirá en la que es actualmente la mejor solución y por el paso 2 todos los pinguinos de ese grupo comenzarán a aproximarse a esta nueva mejor solución.
Es de esta manera que simulamos el paso 3, cada uno de los pasos ahora pueden ser simulados y de esta manera es que discretizamos la heuristica.
Debido a esto es que lo siguiente que realizaremos es realizar una función de costo que permita que estos cambios en los vertices sean guiados a encontrar cada vez una mejor solución. La función de costo es realizada haciendo uso de lo siguiente:

$$w = \begin{cases} w(u,v) \text{ si } (u,v) \in E \\ d(u,v) * maxDiametro(G) * k \text{en otro caso.} \end{cases}$$

Donde d corresponde a la distancia en el $floy - warshall$ donde esta distancia corresponde a la distancia minima conseguible entre los vertices existentes de la gráfica. De la misma manera maxDiametro corresponde al diametro maximo en la grafica original.
Esta función es la que nos guiará en el proceso para conseguir cada vez arboles de menor costo, castigando por sobre manera a aquellas aristas que no se encuentren en la gráfica y permitiendo que aquellas aristas que se encuentren inicialmente en la gráfica sean beneficiadas enormemente para que la heuristica se guie completamente hacía estas aristas.
De esta manera es que se comienza a llenar la función de costo de un arbol, pero además de perjudicar a aquellas aristas que no se encuentran en el árbol, buscamos que la diferencia entre un arbol que tenga todas sus aristas en su grafica original y un árbol que tenga aristas fuera de la grafica original sea notoria. Es por ello que se opto por hacer uso de un normalizador, el cual es el encargado de marcar la diferencia entre un arbol con aristas existentes y uno sin el.
Para ello se trabajo con un normalizador tal que su valor estuviera compuesto con las k aristas de mayor peso en la gráfica original, esto nos mostraría que los arboles que tengan almenos un vertice que no se encuentre en la gráfica original tenga valor mayor que 1 y si tiene vertices inexistentes el valor es menor que 1.
Esto se consigue haciendo que el peso del arbol se divida entre el normalizador. Consiguiendo asi normalizar los datos entre 0 y 1. Bajo este proceso es que podemos evaluar costos de tal forma que es más simple visualizar los arboles que se encuentran en la gráfica original.
Finalmente la implementación principal de la heuristica, queda de la siguiente manera

```
1       pub fn iniciar_PeSOA(&mut self, num_pinguinos: usize, num_grupos:
            usize) {
2         let mut pinguinos = Vec::new();
3         let todos : Vec<usize> = (0..self.grafica.size).collect();
```

```
        for i in 0..num_pinguinos {
            let sol : BTreeSet<usize> = todos.choose_multiple(&mut self.random, self.k).cloned().collect();
            let arbol = self.grafica.arbol_generador_minimo(sol.clone(), *sol.iter().next().unwrap(), self.k);
            let fit = self.calcular_peso(arbol);
            pinguinos.push(Pinguino {
                solucion: sol,
                fitness: fit,
                identificador:i,
            });
        }

        let pinguinos_por_grupo = (num_pinguinos as f64/ num_grupos as f64).ceil() as usize;
        for chunk in pinguinos.chunks(pinguinos_por_grupo) {
            self.grupos.push(GrupoPinguinos {
                pinguinos: chunk.to_vec(),
                mejor_pinguino: None,
            });
        }

        self.actualizar_pesos();
    }

    fn iterar(&mut self, clavados: usize) {
        let normalizador = self.normalizador;

        for grupo in &mut self.grupos {
            for pinguino in &mut grupo.pinguinos {
                for _ in 0..clavados {
                    let mut nueva = pinguino.solucion.clone();
                    if let Some(&vertice_quitar) = pinguino.solucion.iter().collect::<Vec<_>>().choose(&mut self.random) {
                        nueva.remove(&vertice_quitar);
                        let mut vertice_aniadir = None;

                        if self.random.gen_range(0.0..1.0) < 0.2 {
                            if let Some(mejor_local) = &grupo.mejor_pinguino {
                                let candidatos:Vec<usize> = mejor_local.solucion.iter().filter(|&&v| !pinguino.solucion.contains(&v))
                                    .cloned().collect();
                                if !candidatos.is_empty() {
                                    vertice_aniadir = candidatos.choose(&mut self.random).cloned();
                                }
                            }
                        }

                        if vertice_aniadir.is_none() {
                            let candidatos : Vec<usize> = (0..self.grafica.size).filter(|v| !pinguino.solucion.contains(v)).collect();
```

```
49
50                              vertice_aniadir = candidatos.choose(&mut
                                    self.random).copied();
51                          }
52
53                          if let Some(v_aniadir) = vertice_aniadir {
54                              nueva.insert(v_aniadir);
55                          } else {
56                              nueva.insert(*vertice_quitar);
57                          }
58
59                      }
60                      let arbol =
                            self.grafica.arbol_generador_minimo(nueva.clone(),
                            *nueva.iter().next().unwrap(), self.k);
61                      let fit = arbol.peso_arbol_generador / normalizador;
62                      if fit < pinguino.fitness {
63                          pinguino.solucion = nueva;
64                          pinguino.fitness = fit;
65                      }
66                  }
67              }
68          }
69
70          self.actualizar_pesos();
71
72          self.grupos.sort_by(|a,b|
73                              a.mejor_pinguino.as_ref().map_or(f64::INFINITY,
                                    |p| p.fitness)
74                              .partial_cmp(&b.mejor_pinguino.as_ref().map_or(f64::INFI
                                    |p| p.fitness))
75                              .unwrap()
76          );
77
78          if self.grupos.len() > 1 {
79              if let Some(mejor_pinguino_global) =
                    self.mejor_pinguino_actual.clone() {
80                  if let Some(peor_pinguino) =
                        self.grupos[0].pinguinos.iter_mut().max_by(|a,b|
                        a.fitness.partial_cmp(&b.fitness).unwrap()) {
81                      peor_pinguino.solucion =
                            mejor_pinguino_global.solucion;
82                      peor_pinguino.fitness =
                            mejor_pinguino_global.fitness;
83                  }
84              }
85          }
86
87          self.actualizar_pesos();
88      }
```

Con esto es que inicializamos las soluciones en cada uno de los pinguinos, mientras que agruparemos los pinguinos en grupos del mismo tamaño. Esto es lo que inicializa nuestros pinguinos en puntos distintos del espacio de busqueda.

```
1       fn iterar(&mut self, clavados: usize) {
2         let normalizador = self.normalizador;
3
4         for grupo in &mut self.grupos {
```

```
5               for pinguino in &mut grupo.pinguinos {
6                   for _ in 0..clavados {
7                       let mut nueva = pinguino.solucion.clone();
8                       if let Some(&vertice_quitar) =
                            pinguino.solucion.iter().collect::<Vec<_>>().choose(&mut
                            self.random) {
9                           nueva.remove(&vertice_quitar);
10                          let mut vertice_aniadir = None;
11
12                          if self.random.gen_range(0.0..1.0) < 0.2 {
13                              if let Some(mejor_local) =
                                    &grupo.mejor_pinguino {
14                                  let candidatos:Vec<usize> =
                                        mejor_local.solucion.iter().filter(|&&v|
                                        !pinguino.solucion.contains(&v))
15                                      .cloned().collect();
16                                  if !candidatos.is_empty() {
17                                      vertice_aniadir =
                                            candidatos.choose(&mut
                                            self.random).cloned();
18                                  }
19                              }
20                          }
21
22                          if vertice_aniadir.is_none() {
23                              let candidatos : Vec<usize> =
                                    (0..self.grafica.size).filter(|v|
                                    !pinguino.solucion.contains(v)).collect();
24
25                              vertice_aniadir = candidatos.choose(&mut
                                    self.random).copied();
26                          }
27
28                          if let Some(v_aniadir) = vertice_aniadir {
29                              nueva.insert(v_aniadir);
30                          } else {
31                              nueva.insert(*vertice_quitar);
32                          }
33
34                      }
35                      let arbol =
                            self.grafica.arbol_generador_minimo(nueva.clone(),
                            *nueva.iter().next().unwrap(), self.k);
36                      let fit = arbol.peso_arbol_generador / normalizador;
37                      if fit < pinguino.fitness {
38                          pinguino.solucion = nueva;
39                          pinguino.fitness = fit;
40                      }
41                  }
42              }
43          }
44
45          self.actualizar_pesos();
46
47          self.grupos.sort_by(|a,b|
48                              a.mejor_pinguino.as_ref().map_or(f64::INFINITY,
                                    |p| p.fitness)
49                              .partial_cmp(&b.mejor_pinguino.as_ref().map_or(f64::INFI
```

```
                                |p| p.fitness))
                            .unwrap()
        );

        if self.grupos.len() > 1 {
            if let Some(mejor_pinguino_global) =
               self.mejor_pinguino_actual.clone() {
                if let Some(peor_pinguino) =
                   self.grupos[0].pinguinos.iter_mut().max_by(|a,b|
                   a.fitness.partial_cmp(&b.fitness).unwrap()) {
                    peor_pinguino.solucion =
                       mejor_pinguino_global.solucion;
                    peor_pinguino.fitness =
                       mejor_pinguino_global.fitness;
                }
            }
        }

        self.actualizar_pesos();
    }
```

Mientras que iterar es la función que se encarga de los cambios en las soluciones durante cada una de las iteraciones, entre ellas vemos a que corresponde cada uno de los nombres de nuestras variables, vemos que clavados se refiere al numero de veces que nuestro pinguino intentará mejorar su solución actual. De la misma manera contamos con una probabilidad de aceptar a un vertice de la mejor solución, mientras que esta probabilidad inicialmente se encontraba en 60 %, posteriormente se redujo a 20 % debido a que la aproximación de los vertices a la mejor solución era demasiado rápida, ocasionando que los pinguinos lleguen a la mejor solución de una manera tan deprisa que no lográn explorar correctamente, cayendo en un sesgado minimo local.

De esta forma vemos que hacemos uso de la función de actualizar peso, la cual es la siguiente:

```
    fn actualizar_pesos(&mut self) {
      let mut min_global = f64::INFINITY;
      let mut candidato_global = None;
      for grupo in &mut self.grupos {
          let mut min = f64::INFINITY;
          let mut candidato = None;
          for p in &grupo.pinguinos {
              if p.fitness < min {
                  candidato = Some(p.clone());
                  min = p.fitness;
              }
              //println!("Identificador {}, Costo
                 {}",p.identificador, p.fitness);
          }
          grupo.mejor_pinguino = candidato;
          if let Some(mejor_local) = &grupo.mejor_pinguino {
              if mejor_local.fitness < min_global {
                  min_global = mejor_local.fitness;
                  candidato_global = Some(mejor_local.clone());
              }
          }
      }

      self.mejor_pinguino_actual = candidato_global;
      println!("Mejor peso = {}",
```

```
            self.mejor_pinguino_actual.clone().unwrap().fitness);
25      }
```

Función la cual se encarga de actualizar los pesos de los grupos de pinguinos, cambiando el mejor pinguino de cada grupo y el mejor pinguino que se encuentra globalmente en el conjunto de grupos de pinguinos.
Finalmente la ultima función que nos encontramos es correr PeSoa, función la cual es la encargada de correr las iteraciones. La función se encarga de correr los niveles. Niveles los cuales son entendidos como una vez que hemos dejado de mejorar la solución global entonces el nivel aumenta en 1.
Una vez terminado este proceso, se ejecuta un barrido en la mejor solución, el cual se encarga de mejorar la solución actual por fuerza bruta lo más posible hasta que este proceso ya no pueda realizarse más, es decir, hemos llegado a un minimo local.

```
1   pub fn run_pesoa(&mut self,niveles : usize, clavados: usize, epsilon:
        f64) {
2          let mut actual : f64 = 0.0;
3          let mut i = 0;
4          while niveles > i {
5              self.iterar(clavados);
6              if let Some(mejor) = &self.mejor_pinguino_actual {
7                  if (actual - mejor.fitness).abs() < epsilon{
8                      i = i + 1;
9                  }
10                 actual = mejor.fitness;
11                 println!("Iteracion {}: Mejor peso = {}", i ,
                       mejor.fitness);
12             }
13         }
14
15         self.barrido();
16         println!("Solucion postBarrido: {}",
               self.mejor_pinguino_actual.clone().unwrap().fitness);
17      }
```

Siendo de esta forma que hemos concluido las modificaciones pertinentes a la heuristicas.

Resultados

Para la heuristica se han realizado un par de pruebas que nos demuestran la eficacia de la misma, probando en primera instancia con 1000 pinguinos, 50 grupos, 50 niveles y 5 clavados, donde para la mejor solución obtenida usando estos valores para la k de 40 corresponde a: 0,020172468118837713 con la semilla 69.
Posteriormente se ha intentado usando esta configuración para la configuración correspondiente a k igual a 150, donde le mejor resultado obtenido está dado por 0,15776678331840607 para semilla 52. Lo cual después probando me di cuenta que esta solución no es la mejor alcanzable, si no que la heuristica no llega a profundizaar correctamente en los resultados debido a que los niveles son demasiado pequeños.
Aumentar los niveles con la configuración actual no resultaba beneficioso debido a que los tiempos que toma dcon la configuración actual aumentar los niveles resulta perjudicial en los tiempos que toma a cada semilla ser procesada, este aumento de los niveles no fue una opción, por lo que se opto por bajan los pinguinos, aumentar los grupos para mayor exploración y subir el numero de niveles.
De tal manera que tenemos 300 pinguinos, 100 grupos y 100 niveles, pero esta configuración tarda mucho todavia y los resultados no varian demasiado, es por ello que la configuración por

la que finalmente se opto fue opor 120 pinguinos, con 20 grupos, 240 iteraciones y 5 clavados obtenemos el siguiente resultado 0,013484717846826592.
Finalmente es que al hacer el mismo testeo con 240 niveles nos entrega el siguiente resultado 0,05784226550559608 este resultado es optimo, pero vemos que al momento de realizar el barrido, realizamos muchos cambios, es decir, estamos aun lejos del minimo local, por lo que para acercarlo lo que debo realizar es aumentar el tamaño de los niveles a 300 para acercar los resultados aún más al minimo local, dando que de esta manera quedamos de la siguiente manera: 0,05758093801084533
Finalmente con el barrido para k igual a 150 llegamos a 0,05718143633531069.

Conclusión y Trabajos Futuros

Es de esta que podemos concluir que la mejor configuración actual está dada por 120 pinguinos, 20 grupos, 300 niveles, pero esta configuración todavia nos mantiene lejos del minimo local, por lo que se debe de encontrar una manera de acercar mas las soluciones al minimo local sin necesitar del barrido.
Por ello lo más recomendable es optimizar el código de tal manera que tome un menor numero de tiempo para que evalue cada una de las semillas. Esto con la finalidad de poder realizar un mayor numero de niveles en un menor tiempo.
Finalmente otra de las formas de continuar con esta implementación es cambiar la manera en la que discretiza la heuristica, de una manera en la que la penalización de los niveles dependa del grupo en cuestión y no únicamente de todo el conjunto de pinguinos. De tal manera que la propuesta es buscar una implementación que en lugar de en cada iteración acercar a la mejor solución global cada grupo, hacerlo unicamente cuando nos encontramos en un minimo local, esto quiere decir, que el nivel se convertirá en un atributo de grupos y no en un atributo general de la heuristica, penalizando de esta manera el momento en el que el grupo ya no consiga mejores resultados, buscando acercarlo a la mejor solución global en ese momento y no antes.
Hacer esto garantizaría una exploración mejor en sin necesidad de un número de pinguinos tan amplio. Concluyendo de esta manera que a la heuristica le falta mejorar todavía, pero los resultados son optimos aún considernado esto, llegando de esta manera a que los mejores resultados obtenidos son: 0,013484717846826592 para k igual a 40 y 0,05718143633531069 para k igual a 150

Referencias

[1] Gheraibia, Y., Moussaoui, A., Yin, P.-Y., Papadopoulos, Y., & Maazouzi, S. (2018). PeSOA: Penguins Search Optimisation Algorithm for Global Optimisation Problems. *International Arab Journal of Information Technology*, 16(3), 371-384.

[2] Elder, M. (Scribe) & Chawla, S. (Lecturer). (2007). *Lagrangian Relaxation*. CS880: Approximation Algorithms, University of Wisconsin–Madison. Lecture notes from April 26–27, 2007.