JSSP Local Search

Alfonso Fierro López

Week 1, October 2023

Cambios en la estructura del código

Actividades
Descripción y propósito

Descripción

Una actividad a_{tm} es una pareja ordenada (t, m) que asocia un trabajo t con una máquina m. Puede crearse una estructura de orden que permita a cada actividad ser comparada para establecer *presedencia*

Propósito

Abstraer la estructura del problema a la organización de elementos más pequeños permite ampliar la cantidad de soluciones alcanzables mediante algoritmos de búsqueda.

Presedencia y soluciones

Implementación
Definiciones

Los algoritmos propuestos utilizan *listas* para representar los siguientes conceptos:

Solución

$$S = [a_{tm}, a_{jn}, ..., a_{ko}]$$

Donde:

- a_{tm} := Actividad (Trabajo t en máquina m).
- S := Vector ordenado de todas las actividades (orden de procesamiento de las actividades).

Presedencia

$$a_{tm} \prec a_{wo}$$

Se dice que a_{tm} precede a la actividad a_{wo} solo sí

- $t = w$ Son actividades de un mismo trabajo.
- $i(m) < i(o)$. En el programa del trabajo, el índice de la máquina m es menor que el de la máquina o.

1. Actividades de distintos trabajos son incomparables.
2. Actividades que ocupan la posición i en la solución S deben ser incomparables o precedentes a actividades en la posición $j > i$

Vecindarios

Creación de vecindarios

Dada una solución S y una actividad a es posible explorar soluciones próximas o *vecindarios* si se generan nuevas soluciones S' que posiciones distintas y factibles de la actividad a. Para generar estos vecindarios se presentan 3 métodos

Inserción hacia atrás
Elementos

- ¿Cuál es la posición de la actividad en la solución?
- ¿Qué elementos son anteriores a la actividad?
- ¿Cuál es el primer elemento que comparte trabajo con la actividad?

es posible insertar la actividad entre los trabajos anteriores a su posición original y posteriores a la posición de la primer actividad que comparta su respectivo trabajo-

Pseudocódigo
Función Auxiliar-Desplazamiento

```
1: function DESPLAZAR(lista, pos, elementp)
2:     displaced.remove(element)
3:                                   ▷ Inserte elemento en la posición específica
4:     displaced.insert(pos, element)
5:     return displaced
6: end function
```

Pseudocódigo

Función de inserción hacia atrás

Algorithm 8 Inserción hacia atrás

```
1:  function BackInsertion(orden, actividades, actividad)
2:      intercambios ← []
3:      copiar_actividades ← orden.copia()
4:      índice_actividad ← orden.índice(actividad)
5:      cola ← copiar_actividades[: índice_actividad]
6:      act ← actividades[actividad]
7:      for elemento en invertir(cola) do
8:          desplazado ← actividades[elemento]
9:          if desplazado[0] ≠ act[0] then
10:             índice_desplazado ← orden.índice(elemento)
11:             intercambios.agregar(desplazar_posiciones(copiar_actividades, índice_desplazado, actividad))
12:         else
13:             romper
14:         end if
15:     end for
16:     return intercambios
17: end function
```

Figure: Inserción hacia atrás

Inserción hacia adelante
Elementos

- ¿Cuál es la posición de la actividad en la solución?
- ¿Qué elementos son posteriores a la actividad?
- ¿Cuál es el primer elemento (posterior) que comparte trabajo con la actividad?

es posible insertar la actividad entre los trabajos posteriores a su posición original y anteriores a la posición de la primer actividad que comparta su respectivo trabajo.

Pseudocódigo

Función de inserción hacia adelante

Algorithm 9 Inserción hacia adelante

```
1:  function FrwrdInsertion(orden, actividades, actividad)
2:      intercambios ← []
3:      copiar_actividades ← orden.copia()
4:      índice_actividad ← orden.índice(actividad)
5:      cabeza ← copiar_actividades[índice_actividad + 1 :]
6:      act ← actividades[actividad]
7:      for elemento en cabeza do
8:          desplazado ← actividades[elemento]
9:          if desplazado[0] ≠ act[0] then
10:             índice_desplazado ← orden.índice(elemento)
11:             intercambios.agregar(displace_positions(copiar_actividades, índice_desplazado, actividad))
12:         else
13:             romper
14:         end if
15:     end for
16:     return intercambios
17: end function
```

Figure: Inserción hacia adelante

Intercambio de posiciones
Elementos

- ¿Cuál es la posición de la actividad en la solución?
- ¿Qué elementos son anteriores a la actividad?
- ¿Qué elementos son posteriores a la actividad?
- ¿Cuáles son los primeros elementos (posterior y anterior) que comparten trabajo con la actividad?

Puede intentarse intercambiar la actividad entre los trabajos posteriores a la primer actividad anterior que comparta su respectivo trabajo, y anteriores a la posición de la primer actividad posterior que comparta su respectivo trabajo. La factibilidad del intercambio no está garantizada, debe validarse que la solución sea factible en cada movimiento.

Pseudocódigo

Función auxiliar de intercambio

```
1: function SwapPositions(lista, pos1, pos2)
2:     desplazada ← lista.copia()
3:     desplazada[pos1], desplazada[pos2] = desplazada[pos2], desplazada[pos1]
4:     return desplazada
5: end function
```

Intercambio

Función de Intercambio

Algorithm 10 Intercambiar Elementos en el Vecindario

```
1:  function INTERCAMBIAR(orden, actividades, actividad)
2:      intercambios ← []
3:      copiar_actividades ← orden.copia()
4:      índice_actividad ← orden.índice(actividad)
5:      cola ← copiar_actividades[: índice_actividad]
6:      cabeza ← copiar_actividades[índice_actividad + 1 :]
7:      act ← actividades[actividad]
8:      for swapped en invertir(cola) do
9:          swap ← actividades[swapped]
10:         if swap[0] ≠ act[0] then
11:             índice_swap ← orden.índice(swapped)
12:             intercambios.agregar(swap_positions(copiar_actividades, índice_swap, índice_actividad))
13:         else
14:             romper
15:         end if
16:     end for
17:     for swapped en cabeza do
18:         swap ← actividades[swapped]
19:         if swap[0] ≠ act[0] then
20:             índice_swap ← orden.índice(swapped)
21:             intercambios.agregar(swap_positions(copiar_actividades, índice_swap, índice_actividad))
22:         else
23:             romper
24:         end if
25:     end for
26:     return intercambios
27: end function
```

Figure: Función de intercambio

Búsqueda Local

Búsqueda Local

Nociones

Las soluciones propuestas utilizan 2 formas de generar soluciones a partir de una búsqueda local:

- Primer mejora (First improvement)
- Mejor mejora (Best improvement)

Además, se intenta que la mayor parte de los vecindarios se exploren para cada actividad.
Se exploran las 3 maneras de generar vecindarios explicadas anteriormente.
Los criterios de parada son un número de iteraciones y un límite de tiempo.

Primer mejora
First Improvement

Algorithm 11 Búsqueda Local con First Improvement

```
1:  function BusquedaLocalFirstImprovement(solucion_inicial)
2:      solucion_actual ← solucion_inicial
3:      mejor_solucion ← solucion_inicial
4:      while no se cumpla el criterio de parada do
5:          vecindario ← generar_vecindario(solucion_actual)
6:          encontrado ← falso
7:          for cada solucion_vecina en vecindario do
8:              if evaluar(solucion_vecina) < evaluar(solucion_actual) then
9:                  solucion_actual ← solucion_vecina
10:                 encontrado ← verdadero
11:                 romper
12:             end if
13:         end for
14:     end while
15:     return mejor_solucion
16: end function
```

Figure: First Improvement

Mejor mejora

Best Improvement

Algorithm 12 Búsqueda Local con Best Improvement

```
1:  function BusquedaLocalBestImprovement(solucion_inicial)
2:      solucion_actual ← solucion_inicial
3:      mejor_solucion ← solucion_inicial
4:      while no se cumpla el criterio de parada do
5:          vecindario ← generar_vecindario(solucion_actual)
6:          mejor_evaluacion ← evaluar(solucion_actual)
7:          for cada solucion_vecina en vecindario do
8:              evaluacion_vecina ← evaluar(solucion_vecina)
9:              if evaluacion_vecina < mejor_evaluacion then
10:                 solucion_actual ← solucion_vecina
11:                 mejor_evaluacion ← evaluacion_vecina
12:             end if
13:         end for
14:         if mejor_evaluacion < evaluar(mejor_solucion) then
15:             mejor_solucion ← solucion_actual
16:         end if
17:
18:     end while
19:     return mejor_solucion
20: end function
```

Figure: First Improvement

Resultados

Comparación de Algoritmos

Se fijan los parámetros

$$a = 0.7 \quad k = 5$$

El máximo número de iteraciones para los algoritmos aleatorios es 100
El tiempo máximo usado para resolver una instancia está fijo en 1 minuto.

Makespan logrado

	Constructivo	GRASP1	GRASP2	LS1	LS2	LS3	LS4	LS5	LS6
JSSP1.txt	1879	1733	1524	1380	1380	1346	1346	1089	1080
JSSP2.txt	1772	1672	1703	1348	1348	1364	1364	1287	1164
JSSP3.txt	2109	2006	2049	1613	1561	1611	1608	1530	1375
JSSP4.txt	2349	2103	2023	1502	1505	1534	1531	1355	1537
JSSP5.txt	2380	2359	2351	1873	1873	1877	1871	1561	1734
JSSP6.txt	2771	2447	2295	1782	1796	1869	1866	1763	1763
JSSP7.txt	2630	2724	2746	2106	2064	2122	2122	1995	1921
JSSP8.txt	2985	2614	2547	2163	2138	2192	2175	1985	2103
JSSP9.txt	2893	2933	2978	2620	2620	2570	2549	2426	2326
JSSP10.txt	3056	3142	3006	2485	2485	2391	2391	2307	2212
JSSP11.txt	4229	4059	3939	3471	3471	3428	3428	3339	3314
JSSP12.txt	4179	3979	3798	3387	3387	3410	3410	3207	3286
JSSP13.txt	4570	4381	4286	3379	3379	3351	3351	3258	3271
JSSP14.txt	4558	4399	4298	3454	3474	3511	3511	3435	3425
JSSP15.txt	7473	7004	7106	5995	5995	6006	6006	6006	6006
JSSP16.txt	7070	7034	7013	5787	5787	5787	5787	5787	5752

Figure: Makespan logrado para cada instancia

Distancia a la cota inferior

	Constructivo	GRASP1	GRASP2	LS1	LS2	LS3	LS4	LS5	LS6
JSSP1.txt	0.896064581	0.748739	0.537841	0.392533	0.392533	0.358224	0.358224	0.09889	0.089808
JSSP2.txt	0.798984772	0.697462	0.728934	0.368528	0.368528	0.384772	0.384772	0.306599	0.181726
JSSP3.txt	0.79184367	0.704333	0.740867	0.370433	0.326253	0.368734	0.366185	0.299915	0.168224
JSSP4.txt	0.928571429	0.726601	0.66092	0.233169	0.235632	0.259442	0.256979	0.112479	0.261905
JSSP5.txt	0.946034342	0.928863	0.922322	0.53148	0.53148	0.534751	0.529845	0.27637	0.417825
JSSP6.txt	1.181889764	0.926772	0.807087	0.40315	0.414173	0.471654	0.469291	0.388189	0.388189
JSSP7.txt	0.446644664	0.49835	0.510451	0.158416	0.135314	0.167217	0.167217	0.09736	0.056656
JSSP8.txt	0.624047878	0.422198	0.385745	0.176823	0.163221	0.192601	0.183351	0.079978	0.144178
JSSP9.txt	0.542110874	0.563433	0.58742	0.396588	0.396588	0.369936	0.358742	0.293177	0.239872
JSSP10.txt	0.713004484	0.761211	0.684978	0.392937	0.392937	0.340247	0.340247	0.293161	0.23991
JSSP11.txt	0.49699115	0.436814	0.394336	0.228673	0.228673	0.213451	0.213451	0.181947	0.173097
JSSP12.txt	0.392535821	0.325891	0.265578	0.128624	0.128624	0.136288	0.136288	0.068644	0.094968
JSSP13.txt	0.566678094	0.501885	0.469318	0.158382	0.158382	0.148783	0.148783	0.116901	0.121358
JSSP14.txt	0.545084746	0.491186	0.456949	0.170847	0.177627	0.190169	0.190169	0.164407	0.161017
JSSP15.txt	0.352579186	0.267692	0.286154	0.085068	0.085068	0.087059	0.087059	0.087059	0.087059
JSSP16.txt	0.24559549	0.239253	0.235553	0.019556	0.019556	0.019556	0.019556	0.019556	0.01339

Figure: Calidad de las soluciones en comparación a las cotas inferiores

Conclusiones

Conclusiones
Resultados logrados

- El uso de actividades para planificar la solución de las instancias es innegablemente mejor que el uso de trabajos completos.
- Los posibles vecindarios fabricables y evaluables están limitados por la solución planteada inicialmente. Esto es que la solución de las instancias continúa fuertemente ligada a la calidad de la solución inicial
- Es común que cuando la solución de una instancia presente muchos *bloques*, la exploración de los vecindarios no sea capaz de generar mejores soluciones, esto último se intensifica en la inserción hacia adelante, ya que se espera que cada posición adelantada provoque que el *makespan* aumente.

- La razón más común por la que terminan los algoritmos es por el límite de tiempo.