Anexos

Anexo A. Diagrama de Gantt de ejecución de tareas.

Tareas	Inicio	Final	02-Jul-18	09-Jul-18	16-Jul-18	23-Jul-18	30-Jul-18	06-Aug-18	13-Aug-18	20-Aug-18	27-Aug-18	03-Sep-18	10-Sep-18	17-Sep-18	24-Sep-18	01-Oct-18	08-Oct-18	15-Oct-18	22-Oct-18	29-Oct-18	05-Nov-18	12-Nov-18	19-Nov-18	26-Nov-18	03-Dec-18	10-Dec-18	17-Dec-18	24-Dec-18	31-Dec-18	07-Jan-19
Revisión del problema de equilibrado de líneas	02-Jul-18	31-Aug-18																											
Estudio del estado del arte del problema MALBP	01-Sep-18	23-Sep-18																											
Definición del problema	24-Sep-18	14-Oct-18																											
Estudio de la programación en Python	17-Sep-18	15-Oct-18																											
Diseño e implantación de los algoritmos	22-Oct-18	30-Nov-18																											
Testeo de los ejemplares	01-Dec-18	05-Dec-18																											
Análisis de los resultados	06-Dec-18	31-Dec-18																											
Desarrollo de la memoria	01-Sep-18	07-Jan-19																											

Anexo B. Tiempos de ejecución de la experiencia computacional.

Todos los valores mostrados en la siguiente tabla son segundos (s):

PROBLEMA	Tareas	TC	Solución inicial	Optimización local (IMAX=1)	Optimización local (IMAX=2)
Bowman	8	20	0.0556977	0.011612732	0.030175217
Mansoor	11	48	0.0650823	0.005286571	0.071119267
		62	0.0631504	0.008627293	0.041044974
		94	0.0638491	0.003378055	0.062510846
Mertens	7	6	0.0538626	0.006790162	0.006916932
		7	0.0436029	0.008333549	0.009376012
		8	0.0449059	0.008373344	0.008918985
		10	0.0494077	0.001770259	0.014897659
		15	0.0443763	0.001878156	0.016405764
		18	0.0535664	0.002237541	0.037716149
Jaeschke	9	6	0.0651078	0.01382853	0.028972754
		7	0.0680391	0.014671198	0.024739723
		8	0.0685761	0.009875294	0.017637765
		10	0.0572965	0.004948929	0.039721485
		18	0.0568255	0.005788315	0.024348748
Jackson	11	7	0.0823874	0.01400494	0.027693162
		9	0.0729433	0.009220114	0.032563322
		10	0.0752009	0.007734163	0.050608062
		13	0.1036887	0.006441034	0.024937056
		14	0.0717757	0.008737242	0.030013575
		21	0.0700378	0.003373953	0.016246174
Mitchell	21	14	0.2120532	0.050482934	0.440421875
		15	0.1846185	0.235455059	0.448288552
		21	0.1605499	0.087424518	0.5455338
		26	0.1556995	0.029915114	0.444291008
		35	0.1924425	0.027213162	0.26747397
		39	0.1505782	0.019009254	0.411293634
Heskia	28	138	0.233558	0.537365174	1.274383517
		205	0.2274464	0.033779324	0.491322041
		216	0.2185307	0.008333138	0.58064974
		256	0.2183092	0.031962296	0.179156733
		324	0.2237549	0.018756126	0.4225437

		342	0.2289908	0.034678197	0.391396174
Sawyer	30	25	0.2813283	0.187352436	0.916407282
		27	0.2852676	0.214771136	0.646716283
		30	0.2635047	0.301295961	0.55699966
		36	0.2620619	0.072524807	1.610846701
		41	0.2540532	0.059366637	1.059854433
		54	0.2451814	0.037656662	1.791503743
		75	0.2393295	0.048434931	0.773062695
Kilbridge	45	57	0.514806	0.699760451	4.084937951
		79	0.4868441	0.092471498	3.255072314
		92	0.4733897	0.091481959	5.043345284
		110	0.4800502	0.047759238	3.56851393
		138	0.4820634	0.071813422	4.05010878
		184	0.4729286	0.091776523	2.745679277
Tonge	70	176	1.2855872	28.97265063	62.86235101
		364	1.2122854	1.270516846	54.25019682
		410	1.2544249	0.595162989	28.65169721
		468	1.3744103	0.388587555	48.74149548
		527	1.0681273	0.198755116	24.75965509
Arcus-1	83	5048	1.5374878	2.688669155	83.51419715
		5853	1.4437212	1.558948791	87.27839952
		6842	1.417388	1.712640877	103.1021462
		7571	1.4198848	1.28487871	77.41948277
		8412	1.3998602	0.794455952	77.1601085
		8998	1.6244041	1.056029197	69.86857133
		10816	2.2518042	0.711494209	57.68693295
Arcus-2	111	5755	3.0133243	88.29483113	297.965739
		8847	2.5685751	111.0343685	261.9396869
		10027	2.4529322	15.92638842	169.179615
		10743	2.3354252	22.05746428	222.2076868
		11378	2.4482073	9.965922009	254.3941309
		17067	2.3735426	22.36685461	246.5687427

Anexo C. Resultados del algoritmo propuesto (I_{MAX}=1) y soluciones óptimas del problema SALBP.

PROBLEMA	Número de tareas	*c*	SALBP - m	Solución inicial - m	Optimización local - m
Bowman	8	20	5	5	5
Mansoor	11	48	4	4	4
		62	3	3	3
		94	2	2	2
Mertens	7	6	6	7	6
		7	5	5	5
		8	5	5	5
		10	3	4	3
		15	2	3	2
		18	2	2	2
Jaeschke	9	6	8	8	8
		7	7	8	7
		8	6	6	6
		10	4	4	4
		18	3	3	3
Jackson	11	7	8	8	8
		9	6	7	6
		10	5	5	5
		13	4	4	4
		14	4	4	4
		21	3	3	3
Mitchell	21	14	8	10	9
		15	8	9	9
		21	5	7	6
		26	5	5	5
		35	3	4	4
		39	3	4	3
Heskia	28	138	8	9	8
		205	5	6	6
		216	5	5	5
		256	4	5	5
		324	4	4	4
		342	3	4	4
Sawyer	30	25	14	16	14
		27	13	14	14
		30	12	14	13

		36	10	10	10
		41	8	9	9
		54	7	7	7
		75	5	5	5
Kilbridge	45	57	10	12	11
		79	7	10	8
		92	6	9	7
		110	6	6	6
		138	4	5	5
		184	3	4	4
Tonge	70	176	21	24	22
		364	10	12	11
		410	9	9	9
		468	8	9	8
		527	7	7	7
Arcus-1	83	5048	16	18	17
		5853	14	15	14
		6842	12	13	13
		7571	11	11	11
		8412	10	10	10
		8998	9	9	9
		10816	8	8	8
Arcus-2	111	5755	27	30	28
		8847	18	20	19
		10027	16	17	16
		10743	15	16	16
		11378	14	14	14
		17067	9	9	9
TOTAL			**495**	**544**	**517**

Anexo D. Soluciones del algoritmo propuesto a partir de la experiencia computacional y resultados del estudio de Dimitriadis (2006).

			Dimitriadis (2006)		*Algoritmo propuesto (IMAX=2)*					
PROBLEMA	**Tareas**	**TC**	***m***	***p***	***Sol. inicial-m***	***Sol. inicial-p***	***m***	***p***	**Operadores extra**	**Espacio utilizado (%)**
Bowman	8	20	5	5	4	6	4	6	1	80%
Mansoor	11	48	4	4	4	6	3	5	1	75%
		62	3	3	3	5	3	4	1	100%
		94	2	2	2	4	2	3	1	100%
Mertens	7	6	6	6	3	6	3	6	0	50%
		7	5	5	3	6	3	6	1	60%
		8	5	5	3	6	3	6	1	60%
		10	3	3	3	5	3	3	0	100%
		15	2	2	3	3	2	2	0	100%
		18	2	2	2	3	1	2	0	50%
Jaeschke	9	6	8	8	7	9	6	8	0	75%
		7	7	7	6	9	6	7	0	86%
		8	6	6	5	8	5	6	0	83%
		10	4	4	3	6	3	5	1	75%
		18	3	3	2	4	2	4	1	67%
Jackson	11	7	7	8	8	10	6	8	0	86%
		9	5	6	4	8	4	6	0	80%
		10	6	6	4	8	4	6	0	67%
		13	4	4	3	6	3	5	1	75%
		14	3	4	3	6	3	5	1	100%
		21	3	3	2	4	2	4	1	67%
Mitchell	21	14	9	9	8	16	7	8	-1	78%
		15	8	8	8	15	7	8	0	88%
		21	5	5	5	10	5	7	2	100%
		26	5	5	4	8	4	7	2	80%
		35	3	3	4	7	3	5	2	100%
		39	3	3	3	6	3	5	2	100%
Heskia	28	138	6	8	10	14	7	9	1	117%
		205	6	6	6	8	4	7	1	67%
		216	4	5	4	8	4	7	2	100%
		256	5	5	4	7	3	5	0	60%
		324	3	4	3	5	3	5	1	100%

		342	3	3	3	5	3	5	2	100%
Sawyer	30	25	12	14	14	21	11	16	2	92%
		27	13	13	10	17	9	15	2	69%
		30	11	12	9	17	9	14	2	82%
		36	9	10	8	16	8	12	2	89%
		41	8	8	8	15	7	13	5	88%
		54	7	7	6	11	5	9	2	71%
		75	4	5	4	8	4	6	1	100%
Kilbridge	45	57	8	10	10	14	7	12	2	88%
		79	6	7	8	12	6	10	3	100%
		92	5	6	5	10	5	8	2	100%
		110	5	6	5	9	4	7	1	80%
		138	4	4	4	8	4	7	3	100%
		184	3	3	4	6	3	5	2	100%
Tonge	70	176	21	22	18	31	16	26	4	76%
		364	9	10	10	16	8	14	4	89%
		410	7	9	8	15	7	14	5	100%
		468	7	8	6	12	6	12	4	86%
		527	7	7	6	12	6	10	3	86%
Arcus-1	83	5048	16	16	15	26	13	20	4	81%
		5853	13	14	13	23	12	19	5	92%
		6842	10	12	11	21	11	16	4	110%
		7571	11	11	10	18	9	17	6	82%
		8412	10	10	8	16	8	14	4	80%
		8998	8	9	8	15	8	14	5	100%
		10816	8	8	7	12	6	9	1	75%
Arcus-2	111	5755	24	27	21	41	19	31	4	79%
		8847	18	18	17	30	15	24	6	83%
		10027	15	16	14	24	12	20	4	80%
		10743	14	15	13	24	12	21	6	86%
		11378	9	14	11	22	8	18	4	89%
		17067	7	9	8	16	7	14	5	100%
		TOTAL	**462**	**500**	**438**	**775**	**389**	**632**	**2.06**	**85%**

Anexo E. Algoritmos.

Solución inicial:

```
import networkx as nx
import time

lista_completa_solucion = [ ]

def leer_datos(nombre_fichero):
    tiempo_tarea = [ ]
    preced =[ ]
    with open (nombre_fichero, 'r') as f:
        ntareas = int(f.readline().strip())
        for linea in f:
            linea = linea.strip()
            a = eval(linea)
            if type(a) == int :
                tiempo_tarea.append(int(linea))
            elif type(a) != int :
                if linea != '-1,-1' :
                    preced.append(a)
                else:
                    break
    error = len(tiempo_tarea) != ntareas
    return (error, tiempo_tarea, preced)

def crear_grafo (nodos,aristas):
    g= nx.DiGraph()
    id_nodo = 1
    for tiempo_tarea in nodos:
        g.add_node (id_nodo, time=tiempo_tarea, operador = -1, estacion=-1, tiempo_inicio=0, tiempo_acaba=0)
        id_nodo = id_nodo + 1
    g.add_edges_from(aristas)
    return(g)

def nodo_fuente (g):
    inicio_grafo = [ ]
    for nodo in g.node():
        if len (list (g.predecessors(nodo))) > 0 :
            pass
        else:
            inicio_grafo.append(nodo)
    g.add_node(0, time=0, operador =1,estacion=1,tiempo_inicio=0, tiempo_acaba=0)
    for nodo in inicio_grafo:
        g.add_edge (0,nodo)
    return g

def priorizar_tareas_1 (g):
    lista_tareas_ordenadas = []
    sucesores = []
    for i in g.nodes():
        tiempo = g.node[i]['time']
        dic = nx.dfs_successors (g,i)
        for key in dic:
            if len(dic[key]) == 1:
                element = dic[key][0]
                tiempo = tiempo + g.node[element]['time']
            else:
                for element in dic[key]:
                    tiempo = tiempo + g.node [element]['time']
        sucesores.append((i,tiempo))
```

```
    sucesores=sorted (sucesores, key=lambda tup:tup[1])
    for i in sucesores:
        lista_tareas_ordenadas.append(i[0])
    return (lista_tareas_ordenadas)

def comprobar_precedencias (g,tarea):
    for i in g.predecessors(tarea):
        if g.node[i]['operador']==-1:
            return False
        else:
            pass
    return True

def todas_asignadas (g):
    for i in g.nodes():
        if g.nodes[i]['operador'] == -1:
            return False
        else:
            pass
    return True

def crear_operadores_estacion (max_operarios):
    operadores_estacion = []
    i = 1
    while i <= max_operarios:
        operadores_estacion.append (i)
        i = i +1
    return (operadores_estacion)

def abrir_estacion (operadores_estacion,estacion):
    for operador in operadores_estacion:
        lista_completa_solucion.append ([estacion,operador])

def momento_abre_estacion (estacion,tiempo_ciclo):
    momento_apertura_estacion = tiempo_ciclo * (estacion - 1)
    return momento_apertura_estacion

def mirar_cuando_puede_empezar_operador (g,estacion,operador,tiempo_ciclo):
    lista_tareas = [ ]
    lista_aux = [ ]
    for elemento in lista_completa_solucion:
        if elemento[0] == estacion and elemento[1] == operador:
            lista_tareas = elemento [2:]
            for tarea in lista_tareas:
                lista_aux.append (g.node[tarea]['tiempo_acaba'])
    if len(lista_aux) == 0:
        tiempo_empieza_operador = momento_abre_estacion (estacion,tiempo_ciclo)
    else:
        tiempo_empieza_operador = max (lista_aux)
    return tiempo_empieza_operador

def mirar_cuando_puede_empezar_tarea (g,tarea):
    lista_aux = [ ]
    for i in g.predecessors(tarea):
        lista_aux.append (g.node[i]['tiempo_acaba'])
    tiempo_empieza = max(lista_aux)
    return tiempo_empieza

def si_cabe (g,tarea,operador,estacion,tiempo_ciclo):
```

```
    lista_aux = [ ]
    t_proceso = g.node[tarea]['time']
    tiempo_empieza_operador = mirar_cuando_puede_empezar_operador (g,estacion,operador,tiempo_ciclo)
    lista_aux.append (tiempo_empieza_operador)
    tiempo_empieza_tarea = mirar_cuando_puede_empezar_tarea (g,tarea)
    lista_aux.append (tiempo_empieza_tarea)
    tiempo_empieza = max(lista_aux)
    tiempo_disponible = (momento_abre_estacion(estacion,tiempo_ciclo) + tiempo_ciclo) - tiempo_empieza
    if t_proceso <= tiempo_disponible:
        return True
    else:
        False

def escoger_tarea_candidata (g, estacion, lista_tareas_ordenadas, tiempo_ciclo,operadores_estacion):
    for tarea in lista_tareas_ordenadas:
        tarea_candidata = comprobar_precedencias (g,tarea)
        if tarea_candidata == True and g.node[tarea]['operador']== -1:
            for operador in operadores_estacion:
                if si_cabe (g,tarea,operador,estacion,tiempo_ciclo) == True:
                    return tarea
    return None

def escoger_operador (g, estacion, tarea_escogida, tiempo_ciclo, operadores_estacion):
    t_proceso_tarea = g.node[tarea_escogida]['time']
    lista_aux = [ ]
    tiempo_1 = mirar_cuando_puede_empezar_tarea (g,tarea_escogida)
    for operador in operadores_estacion:
        tiempo_2 =mirar_cuando_puede_empezar_operador (g,estacion,operador,tiempo_ciclo)
        lista_aux.append ((operador,tiempo_2))

    lista_aux = sorted (lista_aux, key=lambda tup:tup[1])

    if lista_aux[0][1] >= tiempo_1:
        if (momento_abre_estacion(estacion,tiempo_ciclo) + tiempo_ciclo) - lista_aux[0][1] >= t_proceso_tarea:
            return (lista_aux[0][0],lista_aux[0][1])
        else:
            return (None,None)
    else:
        if (momento_abre_estacion(estacion,tiempo_ciclo) + tiempo_ciclo) - tiempo_1 >= t_proceso_tarea:
            return (lista_aux[0][0],tiempo_1)
        else:
            return (None,None)

def asignar_tarea (g,tarea_escogida,operador_escogido,tiempo_inicio_tarea_escogida,estacion):
    for elemento in lista_completa_solucion:
        if elemento[0] == estacion and elemento[1]==operador_escogido:
            elemento.append(tarea_escogida)
    g.node[tarea_escogida]['operador'] = operador_escogido
    g.node[tarea_escogida]['estacion'] = estacion
    g.node[tarea_escogida]['tiempo_inicio'] = tiempo_inicio_tarea_escogida
    g.node[tarea_escogida]['tiempo_acaba'] = tiempo_inicio_tarea_escogida + g.node[tarea_escogida]['time']

def main(nombre_fichero, tiempo_ciclo, max_operarios):
    error, nodos, aristas = leer_datos(nombre_fichero)
    if error:
        print ('Error en la lectura de datos')
        exit()
    g= crear_grafo(nodos,aristas)
    g= nodo_fuente (g)
    lista_tareas_ordenadas = priorizar_tareas_1(g)
    estacion = 1
    operadores_estacion = crear_operadores_estacion(max_operarios)
    abrir_estacion(operadores_estacion,estacion)
```

```
    finalizar = False
    while not finalizar:
        tarea_escogida = escoger_tarea_candidata(g, estacion, lista_tareas_ordenadas, tiempo_ciclo,
operadores_estacion)
        if tarea_escogida == None:
            estacion = estacion + 1
            abrir_estacion(operadores_estacion,estacion)
        else:
      operador_escogido,tiempo_inicio_tarea_escogida=escoger_operador(g,estacion,tarea_escogida,tiempo_ciclo,
operadores_estacion)
            if operador_escogido == None:
                print("Algo no funciona")
            else:
                asignar_tarea (g,tarea_escogida,operador_escogido,tiempo_inicio_tarea_escogida,estacion)
        finalizar = todas_asignadas(g)
    return (g,lista_completa_solucion)

if __name__ == '__main__':
    main('bowman.txt',20,2)
    main('mansoor.txt',48,2)
    main('mansoor.txt',62,2)
    main('mansoor.txt',94,2)
    main('mertens.txt',6,2)
    main('mertens.txt',7,2)
    main('mertens.txt',8,2)
    main('mertens.txt',10,2)
    main('mertens.txt',15,2)
    main('mertens.txt',18,2)
    main('jaeschke.txt',6,2)
    main('jaeschke.txt',7,2)
    main('jaeschke.txt',8,2)
    main('jaeschke.txt',10,2)
    main('jaeschke.txt',18,2)
    main('jackson.txt',7,2)
    main('jackson.txt',9,2)
    main('jackson.txt',10,2)
    main('jackson.txt',13,2)
    main('jackson.txt',14,2)
    main('jackson.txt',21,2)
    main('mitchell.txt',14,2)
    main('mitchell.txt',15,2)
    main('mitchell.txt',21,2)
    main('mitchell.txt',26,2)
    main('mitchell.txt',35,2)
    main('mitchell.txt',39,2)
    main('heskia.txt',138,2)
    main('heskia.txt',205,2)
    main('heskia.txt',216,2)
    main('heskia.txt',256,2)
    main('heskia.txt',324,2)
    main('heskia.txt',342,2)
    main('sawyer.txt',25,2)
    main('sawyer.txt',27,2)
    main('sawyer.txt',30,2)
    main('sawyer.txt',36,2)
    main('sawyer.txt',41,2)
    main('sawyer.txt',54,2)
    main('sawyer.txt',75,2)
    main('kilbridge.txt',57,2)
```

```
main('kilbridge.txt',79,2)
main('kilbridge.txt',92,2)
main('kilbridge.txt',110,2)
main('kilbridge.txt',138,2)
main('kilbridge.txt',184,2)
main('tonge.txt',176,2)
main('tonge.txt',364,2)
main('tonge.txt',410,2)
main('tonge.txt',468,2)
main('tonge.txt',527,2)
main('arc1.txt',5048,2)
main('arc1.txt',5853,2)
main('arc1.txt',6842,2)
main('arc1.txt',7571,2)
main('arc1.txt',8412,2)
main('arc1.txt',8998,2)
main('arc1.txt',10816,2)
main('arc2.txt',5755,2)
main('arc2.txt',8847,2)
main('arc2.txt',10027,2)
main('arc2.txt',10743,2)
main('arc2.txt',11378,2)
main('arc2.txt',17067,2)
```

Heurística 1:

```
def priorizar_tareas_1 (g):
    lista_tareas_ordenadas = []
    sucesores = []
    for i in g.nodes():
        tiempo = g.node[i]['time']
        dic = nx.dfs_successors (g,i)
        for key in dic:
            if len(dic[key]) == 1:
                element = dic[key][0]
                tiempo = tiempo + g.node[element]['time']
            else:
                for element in dic[key]:
                    tiempo = tiempo + g.node [element]['time']
        sucesores.append((i,tiempo))
    sucesores=sorted (sucesores, key=lambda tup:tup[1])
    for i in sucesores:
        lista_tareas_ordenadas.append(i[0])
    return (lista_tareas_ordenadas)
```

Heurística 2:

```
def priorizar_tareas_2 (g):
    lista_tareas_ordenadas = []
    lista_tareas = []
    for i in g.nodes():
        lista_tareas.append((i,g.node[i]['time']))
    lista_tareas=sorted(lista_tareas, key=lambda tup:tup[1], reverse=True)
    for i in lista_tareas:
        lista_tareas_ordenadas.append(i[0])
    return (lista_tareas_ordenadas)
```

Heurística 3:

```
def priorizar_tareas_3 (g):
    lista_tareas_ordenadas = []
    lista_tareas = []
    for i in g.nodes():
        sucesores=0
        dic=nx.dfs_successors(g,i)
        for key in dic:
```

```
            sucesores = sucesores + len(dic[key])
        lista_tareas.append ((i,sucesores))
    lista_tareas=sorted(lista_tareas, key=lambda tup:tup[1], reverse=True)
    for i in lista_tareas:
        lista_tareas_ordenadas.append(i[0])
    return (lista_tareas_ordenadas)
```

Heurística 4:

```
def priorizar_tareas_4 (g):
    lista_tareas_ordenadas = []
    lista_tareas = []
    for i in g.nodes():
        n_sucesores = len(list (g.successors(i)))
        lista_tareas.append((i,n_sucesores))
    lista_tareas=sorted(lista_tareas, key=lambda tup:tup[1], reverse=True)
    for i in lista_tareas:
        lista_tareas_ordenadas.append(i[0])
    return (lista_tareas_ordenadas)
```

Heurística 5:

```
def priorizar_tareas_5 (g,tiempo_ciclo,max_operarios):
    h=nx.reverse_view(g)
    lista_tareas_ordenadas = []
    lista_LB = []
    for i in h.nodes():
        tiempo = h.node[i]['time']
        dic = nx.dfs_successors (h,i)
        for key in dic:
            if len(dic[key]) == 1:
                element = dic[key][0]
                tiempo = tiempo + h.node[element]['time']
            else:
                for element in dic[key]:
                    tiempo = tiempo + h.node [element]['time']
        formula = float (tiempo)/float(tiempo_ciclo * max_operarios)
        lista_LB.append((i,formula))
    lista_LB = sorted(lista_LB, key=lambda tup:tup[1])
    for i in lista_LB:
        lista_tareas_ordenadas.append(i[0])
    return (lista_tareas_ordenadas)
```

Heurística 6:

```
def priorizar_tareas_6 (g,tiempo_ciclo,max_operarios):
    lista_tareas_ordenadas = []
    lista_UB = []
    n = g.number_of_nodes() -1 #Restamos el nodo fuente
    for i in g.nodes():
        tiempo = g.node[i]['time']
        dic = nx.dfs_successors (g,i)
        for key in dic:
            if len(dic[key]) == 1:
                element = dic[key][0]
                tiempo = tiempo + g.node[element]['time']
            else:
                for element in dic[key]:
                    tiempo = tiempo + g.node [element]['time']
        formula = n + 1 - float (tiempo)/float(tiempo_ciclo * max_operarios)
        lista_UB.append((i,formula))
    lista_UB = sorted(lista_UB, key=lambda tup:tup[1])
```

```
    for i in lista_UB:
        lista_tareas_ordenadas.append(i[0])
    return (lista_tareas_ordenadas)
```

Heurística 7:

```
def priorizar_tareas_7 (g,tiempo_ciclo,max_operarios):
    lista_tareas_ordenadas = []
    lista_tareas_UB = []
    lista_tareas_LB = []
    lista_aux = []
    n= g.number_of_nodes() -1
    for i in g.nodes():
        tiempo = g.node[i]['time']
        dic = nx.dfs_successors (g,i)
        for key in dic:
            if len(dic[key]) == 1:
                element = dic[key][0]
                tiempo = tiempo + g.node[element]['time']
            else:
                for element in dic[key]:
                    tiempo = tiempo + g.node [element]['time']
        formula = n + 1 - (float (tiempo)/float(tiempo_ciclo * max_operarios))
        lista_tareas_UB.append((i,formula))
    h= nx.reverse_view(g)
    for i in h.nodes():
        tiempo = h.node[i]['time']
        dic = nx.dfs_successors (h,i)
        for key in dic:
            if len(dic[key]) == 1:
                element = dic[key][0]
                tiempo = tiempo + h.node[element]['time']
            else:
                for element in dic[key]:
                    tiempo = tiempo + h.node [element]['time']
        formula = float (tiempo)/float(tiempo_ciclo * max_operarios)
        lista_tareas_LB.append((i,formula))
    for i in lista_tareas_LB:
        for j in lista_tareas_UB:
            if i[0]==j[0]:
                lista_aux.append ((i[0], i[1]-j[1]))
                break
            else:
                pass
    lista_aux = sorted (lista_aux, key=lambda tup:tup[1])  e
    for i in lista_aux:
        lista_tareas_ordenadas.append(i[0])
    return(lista_tareas_ordenadas)
```

Heurística 8:

```
def priorizar_tareas_8 (g):
    lista_tareas_ordenadas = []
    for i in g.node:
        lista_tareas_ordenadas.append(i)
    lista_tareas_ordenadas = sorted(lista_tareas_ordenadas)
    return (lista_tareas_ordenadas)
```

Heurística 9:

```
def priorizar_tareas_9 (g):
    lista_tareas_ordenadas = []
    lista_tareas = []
    for i in g.nodes():
        n_sucesores = 0
        tiempo = g.node[i]['time']
```

```
        dic = nx.dfs_successors (g,i)
        for key in dic:
            n_sucesores = n_sucesores + len(dic[key])
            if len(dic[key]) == 1:
                element = dic[key][0]
                tiempo = tiempo + g.node[element]['time']
            else:
                for element in dic[key]:
                    tiempo = tiempo + g.node [element]['time']
        formula = float(tiempo) / float(n_sucesores + 1)
        lista_tareas.append((i,formula))
    lista_tareas=sorted (lista_tareas, key=lambda tup:tup[1], reverse=True)
    for i in lista_tareas:
        lista_tareas_ordenadas.append(i[0])
    return (lista_tareas_ordenadas)
```

Heurística 10:

```
def priorizar_tareas_10 (g,tiempo_ciclo,max_operarios):
    lista_tareas_ordenadas = []
    lista_tareas = []
    n = g.number_of_nodes() -1
    for i in g.nodes():
        n_sucesores = 0
        tiempo = g.node[i]['time']
        dic = nx.dfs_successors (g,i)
        for key in dic:
            n_sucesores = n_sucesores + len(dic[key])
            if len(dic[key]) == 1:
                element = dic[key][0]
                tiempo = tiempo + g.node[element]['time']
            else:
                for element in dic[key]:
                    tiempo = tiempo + g.node [element]['time']
        formula = (n + 1 - (float (tiempo)/float(tiempo_ciclo * max_operarios)))/float(n_sucesores+1)
        lista_tareas.append((i,formula))
    lista_tareas = sorted(lista_tareas, key=lambda tup:tup[1])
    for i in lista_tareas:
        lista_tareas_ordenadas.append(i[0])
    return (lista_tareas_ordenadas)
```

Heurística 11:

```
def priorizar_tareas_11 (g,tiempo_ciclo,max_operarios):
    lista_tareas_ordenadas = []
    lista_tareas = []
    n = g.number_of_nodes() -1
    for i in g.nodes():
        tiempo = g.node[i]['time']
        dic = nx.dfs_successors (g,i)
        for key in dic:
            if len(dic[key]) == 1:
                element = dic[key][0]
                tiempo = tiempo + g.node[element]['time']
            else:
                for element in dic[key]:
                    tiempo = tiempo + g.node [element]['time']
        formula = (float(g.node[i]['time']))/(n + 1 - (float (tiempo)/float(tiempo_ciclo * max_operarios)))
        lista_tareas.append((i,formula))
    lista_tareas = sorted(lista_tareas, key=lambda tup:tup[1], reverse=True)
    for i in lista_tareas:
```

```
        lista_tareas_ordenadas.append(i[0])
    return (lista_tareas_ordenadas)
```

Heurística 12:

```
def priorizar_tareas_12 (g,tiempo_ciclo,max_operarios):
    h=nx.reverse_view(g)
    lista_tareas_ordenadas = []
    lista_tareas = []
    lista_aux= []
    n= g.number_of_nodes() -1
    for i in g.nodes():
        n_sucesores = 0
        tiempo = g.node[i]['time']
        dic = nx.dfs_successors (g,i)
        for key in dic:
            n_sucesores  = n_sucesores + len(dic[key])
            if len(dic[key]) == 1:
                element = dic[key][0]
                tiempo = tiempo + g.node[element]['time']
            else:
                for element in dic[key]:
                    tiempo = tiempo + g.node [element]['time']
        formula = n + 1 - (float (tiempo)/float(tiempo_ciclo * max_operarios))
        lista_tareas.append([i,formula,n_sucesores])
    for i in h.nodes():
        tiempo = h.node[i]['time']
        dic = nx.dfs_successors (h,i)
        for key in dic:
            if len(dic[key]) == 1:
                element = dic[key][0]
                tiempo = tiempo + h.node[element]['time']
            else:
                for element in dic[key]:
                    tiempo = tiempo + h.node [element]['time']
        formula = float (tiempo)/float(tiempo_ciclo * max_operarios)
        for j in lista_tareas:
            if j[0] == i:
                j.append(formula)
                break
            else:
                pass
    for v in lista_tareas:
        lista_aux.append ((v[0],v[2]/(v[1]-v[3])))
    lista_aux = sorted (lista_aux, key=lambda tup:tup[0], reverse=True)
    for i in lista_aux:
        lista_tareas_ordenadas.append (i[0])
    return (lista_tareas_ordenadas)
```

Optimización Local:

```
import networkx as nx
import time
import solucion_inicial

def buscar_huecos(lista,lista_tareas_virtuales):
    for i in range(len(lista)-1):
        if lista[i+1][0]-lista[i][1] != 0:
            lista_tareas_virtuales.append((lista[i][1],lista[i+1][0]))
    return lista_tareas_virtuales

def eliminar_nodos_virtuales_ant (g,nodos_reales):
    lista_nodos = list(g.nodes)
```

```
    for i in nodos_reales:
        if i in lista_nodos:
            lista_nodos.remove(i)
    if len (lista_nodos)>0:
        for nodo_virtual in lista_nodos:
            g.remove_node(nodo_virtual)

def buscar_tareas_virtuales (g,elemento, tiempo_ciclo):
    lista_aux = []
    lista_tareas_virtuales = []
    estacion = elemento[0]
    operador = elemento [1
    tareas = elemento[2:]
    if len(tareas)>0:
        for tarea in tareas:
            lista_aux.append ((g.node[tarea]['tiempo_inicio'],g.node[tarea]['tiempo_acaba']))
        ultima_tarea = len(lista_aux)-1
        lista_aux = sorted (lista_aux, key=lambda tup:tup[0])
        if lista_aux[0][0] != solucion_inicial.momento_abre_estacion (estacion,tiempo_ciclo):
            lista_tareas_virtuales.append (( solucion_inicial.momento_abre_estacion
(estacion,tiempo_ciclo),lista_aux[0][0]))
        lista_tareas_virtuales = buscar_huecos(lista_aux,lista_tareas_virtuales)
        if lista_aux[ultima_tarea][1] !=
(solucion_inicial.momento_abre_estacion(estacion,tiempo_ciclo)+tiempo_ciclo):
            lista_tareas_virtuales.append
((lista_aux[ultima_tarea][1],solucion_inicial.momento_abre_estacion(estacion,tiempo_ciclo)+tiempo_ciclo ))

    return (lista_tareas_virtuales,estacion,operador)

def crear_nodo_virtual (g,estacion,operador,lista_tareas_virtuales):
    id_node = g.number_of_nodes() + 1
    for tarea in lista_tareas_virtuales:
        g.add_node(id_node,time= tarea[1]-tarea[0],estacion= estacion, operador = operador, tiempo_inicio =
tarea[0], tiempo_acaba = tarea[1])
        id_node = id_node + 1

def crear_tareas_virtuales (g,sol_inicial,tiempo_ciclo,nodos_reales):
    eliminar_nodos_virtuales_ant(g,nodos_reales)
    for elemento in sol_inicial:
        lista_tareas_virtuales,estacion,operador = buscar_tareas_virtuales(g,elemento,tiempo_ciclo)
        if len(lista_tareas_virtuales)>0:
            crear_nodo_virtual(g,estacion,operador,lista_tareas_virtuales)
    num_nodos_virtuales = g.number_of_nodes()
    nodos_virtuales = list(range(len(nodos_reales)+1,num_nodos_virtuales+1))
    return (nodos_virtuales)

def identificar_num_operadores (lista_solucion):
    lista_aux=[]
    for elemento in lista_solucion:
        if len(elemento)>2:
            lista_aux.append(elemento)
    num_operadores = len (lista_aux)
    return num_operadores

def si_cabe_real_virtual (g,nodo_real,nodo_virtual,tiempo_inicio):
    if tiempo_inicio + g.node[nodo_real]['time'] <= g.node[nodo_virtual]['tiempo_acaba'] :
        return True
    else:
        return False
```

```
def mirar_predecesores (g,nodo_real,nodo_virtual):
    tiempo_acaba_predecesores = []
    predecesores= list(nx.ancestors(g,nodo_real))
    if len(predecesores)>0:
        for p in predecesores:
            tiempo_acaba_predecesores.append(g.node[p]['tiempo_acaba'])
        if max(tiempo_acaba_predecesores) + g.node[nodo_real]['time'] <= g.node[nodo_virtual]['tiempo_acaba']:
            if max(tiempo_acaba_predecesores) >= g.node[nodo_virtual]['tiempo_inicio']:
                return True, max(tiempo_acaba_predecesores)
            else:
                return True, g.node[nodo_virtual]['tiempo_inicio']
        else:
            return False, None
    else:
        return True,g.node[nodo_virtual]['tiempo_inicio']

def mirar_sucesores (g,nodo_real,nodo_virtual,tiempo_inicio):
    tiempo_inicio_sucesores = []
    sucesores = list(nx.descendants(g,nodo_real))
    if len(sucesores)>0:
        for s in sucesores:
            tiempo_inicio_sucesores.append(g.node[s]['tiempo_inicio'])
        if min (tiempo_inicio_sucesores) >= tiempo_inicio + g.node[nodo_real]['time']:
            return True
        else:
            return False
    else:
        return True

def comprobar_intercambio(g,nodo_real,nodo_virtual):
    estado, tiempo_inicio = mirar_predecesores(g,nodo_real,nodo_virtual)
    if estado == True:
        resultado = mirar_sucesores(g,nodo_real,nodo_virtual,tiempo_inicio)
        if resultado == True:
            cabe = si_cabe_real_virtual(g,nodo_real,nodo_virtual,tiempo_inicio)
            if cabe== True:
                return True,tiempo_inicio
            else:
                return False, None
        else:
            return False, None
    else:
        return False, None

def realizar_intercambio (g,nodo_real,nodo_virtual,tiempo_inicio):
    g.node[nodo_real]['estacion'] = g.node[nodo_virtual]['estacion']
    g.node[nodo_real]['operador']= g.node[nodo_virtual]['operador']
    g.node[nodo_real]['tiempo_inicio']= tiempo_inicio
    g.node[nodo_real]['tiempo_acaba']= tiempo_inicio + g.node[nodo_real]['time']

def eliminar_tarea_intercambiada (g,nodo_real,sol_inicial):
    nueva_lista = [x[:]for x in sol_inicial]
    estacion = g.node[nodo_real]['estacion']
    operador = g.node[nodo_real]['operador']
    for elemento in nueva_lista:
        if elemento[0]==estacion and elemento[1]==operador:
            lista_elem_tareas = elemento[2:]
            indice_tarea_eliminar = lista_elem_tareas.index(nodo_real) + 2
            elemento.pop(indice_tarea_eliminar)
    return (nueva_lista)

def actualizar_lista_solucion (g,nodo_real,nodo_virtual,sol_inicial):
```

```
    lista_sin_tarea = eliminar_tarea_intercambiada (g,nodo_real,sol_inicial)
    estacion = g.node[nodo_virtual]['estacion']
    operador = g.node[nodo_virtual]['operador']

    for elemento in lista_sin_tarea:
        if elemento[0]==estacion and elemento[1]==operador:
            elemento.append (nodo_real)
    return (lista_sin_tarea)

def identificar_num_estaciones (lista_solucion):
    lista_aux= []
    for elemento in lista_solucion:
        if len(elemento)>2:
            lista_aux.append (elemento[0])
    lista_aux = set(lista_aux)
    num_estaciones = len (lista_aux)
    return (num_estaciones)

def identificar_carga_estaciones (g,lista_solucion):
    lista_aux = []
    lista_aux2 = []
    for elemento in lista_solucion:
        if len(elemento)>2:
            lista_aux.append(elemento[2:])
    for i in lista_aux:
        sum_tareas_op = 0
        for tarea in i:
            sum_tareas_op = sum_tareas_op + g.node[tarea]['time']
        lista_aux2.append (sum_tareas_op)
    suma_cuadrados= 0
    for j in lista_aux2:
        suma_cuadrados = suma_cuadrados + (j*j)
    return (suma_cuadrados)

def mejora (lista_mejor_vecino,lista_nuevo_vecino,g):
    if mejorar_vecino_operarios (lista_mejor_vecino,lista_nuevo_vecino) == True:
        return True
    else:
        if mejorar_vecino_estaciones (lista_mejor_vecino,lista_nuevo_vecino)== True:
            return True
        else:
            if mejorar_vecino_carga_estaciones(lista_mejor_vecino,lista_nuevo_vecino,g) == True:
                return True
            else:
                return False

def mejorar_vecino_operarios (lista_mejor_vecino,lista_nuevo_vecino):
    op_mejor_vecino = identificar_num_operadores(lista_mejor_vecino)
    op_nuevo_vecino = identificar_num_operadores(lista_nuevo_vecino)

    if op_nuevo_vecino < op_mejor_vecino:
        return True
    else:
        return False

def mejorar_vecino_estaciones (lista_mejor_vecino,lista_nuevo_vecino):
    estaciones_mejor_vecino = identificar_num_estaciones(lista_mejor_vecino)
    estaciones_nuevo_vecino = identificar_num_estaciones(lista_nuevo_vecino)
```

```
    if estaciones_nuevo_vecino < estaciones_mejor_vecino:
        return True
    else:
        return False

def mejorar_vecino_carga_estaciones (lista_mejor_vecino,lista_nuevo_vecino,g):
    carga_est_mejor_vecino = identificar_carga_estaciones(g,lista_mejor_vecino)
    carga_est_nuevo_vecino = identificar_carga_estaciones(g,lista_nuevo_vecino)

    if carga_est_nuevo_vecino > carga_est_mejor_vecino:
        return True
    else:
        return False

def obtener_vecino_cediendo_tareas (g,nodos_reales,nodos_virtuales,lista_sol_inicial):
    lista_mejor_vecino = lista_sol_inicial
    for nodo_real in nodos_reales:
        for nodo_virtual in nodos_virtuales:
            resultado,tiempo_inicio = comprobar_intercambio(g,nodo_real,nodo_virtual)
            if resultado == True:
                lista_nuevo_vecino = actualizar_lista_solucion(g,nodo_real,nodo_virtual,lista_mejor_vecino)
                if mejora (lista_mejor_vecino,lista_nuevo_vecino,g) == True:
                    lista_mejor_vecino = lista_nuevo_vecino
                    realizar_intercambio (g,nodo_real,nodo_virtual,tiempo_inicio)
                    break
        else:
            continue
        break
    return lista_mejor_vecino

def empieza_antes (g,nodo_real_1,nodo_real_2):
    if g.node[nodo_real_1]['tiempo_inicio']<g.node[nodo_real_2]['tiempo_inicio']:
        return True
    else:
        return False

def comprobar_precedencia (g,nodo_real_1,nodo_real_2):
    lista_pred_2 = list(nx.ancestors(g,nodo_real_2))
    if nodo_real_1 in lista_pred_2:
            return False
    else:
        return True

def comprobar_precedentes (g,nodo_real_1,nodo_real_2):
    lista_precedentes_acaba= []
    lista_precedentes = list(nx.ancestors(g,nodo_real_2))
    for p in lista_precedentes:
        lista_precedentes_acaba.append (g.node[p]['tiempo_acaba'])
    if len (lista_precedentes_acaba) > 0:
        predecesor_acaba_mas_tarde = max (lista_precedentes_acaba)
        if predecesor_acaba_mas_tarde <= g.node[nodo_real_1]['tiempo_inicio']:
            return True
        else:
            return False
    else:
        return True

def comprobar_sucesores (g,nodo_real_1,nodo_real_2):
    lista_sucesores_inicio = []
    lista_sucesores = list(nx.descendants(g,nodo_real_1))
    for s in lista_sucesores:
        lista_sucesores_inicio.append (g.node[s]['tiempo_inicio'])
    if len (lista_sucesores_inicio)>0:
```

```
        sucesor_empieza_antes = min(lista_sucesores_inicio)
        if sucesor_empieza_antes >= g.node[nodo_real_2]['tiempo_acaba']:
            return True
        else:
            return False
    else:
        return True

def comprobar_posterioridad (g,nodo_real_1,nodo_real_2):
    lista_suc_2 = list(nx.ancestors(g,nodo_real_2))
    if nodo_real_1 in lista_suc_2:
        return False
    else:
        return True

def comprobar_intercambio_2 (g,nodo_real_1,nodo_real_2):
    if empieza_antes (g,nodo_real_1,nodo_real_2) == True:
        if comprobar_precedencia (g,nodo_real_1,nodo_real_2) == True:
            if comprobar_precedentes(g,nodo_real_1,nodo_real_2) == True:
                if comprobar_sucesores (g,nodo_real_1,nodo_real_2)== True:
                    return True
    else:
        if comprobar_posterioridad (g,nodo_real_1,nodo_real_2) == True:
            if comprobar_precedentes (g,nodo_real_2,nodo_real_1)== True:
                if comprobar_sucesores (g,nodo_real_2,nodo_real_1) == True:
                    return True


def realizar_intercambio_2 (g,nodo_real_1,nodo_real_2,lista_solucion):
    g.node[nodo_real_1]['estacion'] = g.node[nodo_real_2]['estacion']
    g.node[nodo_real_2]['estacion'] = g.node[nodo_real_1]['estacion']

    g.node[nodo_real_1]['operario'] = g.node[nodo_real_2]['operario']
    g.node[nodo_real_2]['operario'] = g.node[nodo_real_1]['operario']

    g.node[nodo_real_1]['tiempo_inicio'] = g.node[nodo_real_2]['tiempo_inicio']
    g.node[nodo_real_2]['tiempo_inicio'] = g.node[nodo_real_1]['tiempo_inicio']

    g.node[nodo_real_1]['tiempo_acaba'] = g.node[nodo_real_1]['tiempo_inicio'] + g.node[nodo_real_1]['time']
    g.node[nodo_real_12]['tiempo_acaba'] = g.node[nodo_real_2]['tiempo_inicio'] + g.node[nodo_real_2]['time']

def actualizar_lista_solucion_intercambio (g,nodo_real_1,nodo_real_2,sol_inicial):
    lista_sin_nodo_1 = eliminar_tarea_intercambiada (g,nodo_real_1,sol_inicial)
    lista_sin_nodos_interc = eliminar_tarea_intercambiada(g,nodo_real_2,lista_sin_nodo_1)

    estacion_1 = g.node[nodo_real_1]['estacion']
    operador_1 = g.node[nodo_real_1]['operador']

    estacion_2 = g.node[nodo_real_2]['estacion']
    operador_2 = g.node[nodo_real_2]['operador']

    for elemento in lista_sin_nodos_interc:
        if elemento[0]==estacion_1 and elemento[1]==operador_1:
            elemento.append (nodo_real_1)

    for elemento in lista_sin_nodos_interc:
        if elemento[0]==estacion_2 and elemento[1]==operador_2:
            elemento.append (nodo_real_2)
```

```
    return (lista_sin_nodos_interc)

def obtener_vecino_intercambio (g,nodos_reales,solucion_ceder,num_nodos_reales):
    nodos_reales.pop()
    lista_mejor_vecino = solucion_ceder
    for nodo_real_1 in nodos_reales:
        nodos_reales_2 = list(range(nodo_real_1 + 1, num_nodos_reales+1))
        for nodo_real_2 in nodos_reales_2:
            if comprobar_intercambio_2(g,nodo_real_1,nodo_real_2)==True:
                lista_nuevo_vecino =
actualizar_lista_solucion_intercambio(g,nodo_real_1,nodo_real_2,lista_mejor_vecino)
                if mejora(lista_mejor_vecino,lista_nuevo_vecino,g) == True:
                    realizar_intercambio_2(g,nodo_real_1,nodo_real_2,lista_mejor_vecino)
                    lista_mejor_vecino = lista_nuevo_vecino
                    break
        else:
            continue
        break
    return (lista_mejor_vecino)

def OPTIMIZACION_LOCAL_INTERCAMBIO (g,solucion_ceder,tiempo_ciclo,nodos_reales,num_nodos_reales):
    mejorar= True
    solucion_actual = solucion_ceder
    while mejora == True:
        vecino = obtener_vecino_intercambio(g,nodos_reales,solucion_actual,num_nodos_reales)
        if mejorar (solucion_actual,vecino,g) == True:
            solucion_actual = vecino
        else:
            mejorar = False
    return solucion_actual


def OPTIMIZACION_LOCAL_CEDER (g,lista_sol_inicial,tiempo_ciclo,nodos_reales):
    mejorar = True
    solucion_actual = lista_sol_inicial
    while mejorar == True:
        nodos_virtuales = crear_tareas_virtuales (g,solucion_actual,tiempo_ciclo,nodos_reales)
        vecino = obtener_vecino_cediendo_tareas(g,nodos_reales,nodos_virtuales,solucion_actual)
        if mejora (solucion_actual,vecino,g) == True:
          solucion_actual = vecino
        else:
            mejorar = False
    return (solucion_actual)

def optimizacion_local (g,lista_solucion,tiempo_ciclo,nodos_reales,num_nodos_reales):
    start = time.clock()
    solucion_actual = lista_solucion
    mejorar = True
    while mejorar == True:
        solucion_actual = OPTIMIZACION_LOCAL_CEDER(g,lista_solucion,tiempo_ciclo,nodos_reales)
        solucion_intercambio =
OPTIMIZACION_LOCAL_INTERCAMBIO(g,solucion_actual,tiempo_ciclo,nodos_reales,num_nodos_reales)
        if mejora (solucion_actual,solucion_intercambio,g) == True:
            solucion_actual=solucion_intercambio
        else:
            mejorar = False
    eliminar_nodos_virtuales_ant(g,nodos_reales)
    return (solucion_actual,start)

def llamar_mejor_solucion_inicial (nombre_fichero,tiempo_ciclo,max_operarios):
    lista_sol_inicial, g = solucion_inicial.mejor_solucion(nombre_fichero,tiempo_ciclo,max_operarios)
    Error,nodos,aristas = solucion_inicial.leer_datos(nombre_fichero)
    g.remove_node(0) # Quitamos el nodo 0 (nodo fuente)
```

```
    num_nodos_reales = len(nodos)
    nodos_reales = list(range(1,num_nodos_reales+1))
    return (lista_sol_inicial, g , num_nodos_reales,nodos_reales)


def main (nombre_fichero,tiempo_ciclo,max_operarios):
    fichero = open('resultados.txt','a')
    lista_sol_inicial,g,num_nodos_reales,nodos_reales =
llamar_mejor_solucion_inicial(nombre_fichero,tiempo_ciclo,max_operarios)
    solucion_opt,start=optimizacion_local(g,lista_sol_inicial,tiempo_ciclo,nodos_reales,num_nodos_reales)
    end = time.clock()-start
    operarios = identificar_num_operadores(solucion_opt)
    estaciones = identificar_num_estaciones (solucion_opt)
    print (estaciones,operarios,end)
    print (solucion_opt)
    print(g.nodes.data())
    fichero.write(nombre_fichero + '*' + str(tiempo_ciclo) +'*'+str(estaciones) + '*' + str(operarios) + '*' + str(end) +
'\n')
    fichero.close()

if __name__ == '__main__':
    main('bowman.txt',20,2)
    ...
```