An algorithm framework for the service area problem

Dr. Yunfeng Kong

College of Environment and Planning, Henan University, Kaifeng, China

yfkong@henu.edu.cn

1. The service area problem (SAP)

The design of service areas is one of the essential issues in providing efficient services in both the public and private sectors. The delineation of service areas for schools, hospitals, healthcare centers, disaster shelters, and many other facilities can be generalized as a contiguity-constrained capacitated facility SAP. For a geographic area with basic areal units, the SAP should assign the service-demand areal units to service-supply units such that each facility has a service area and some criteria are satisfied. The basic criteria for the problem include the balance of service demand and supply, the highest service accessibility, the contiguity of service areas, and the integrity of basic units. The balance criterion means that the total service demand in each area should be no greater than its service capacity. Service accessibility can be evaluated by total travel distance from demand units to their supply units. The shortest travel distance is usually preferred when using the service. Contiguous service area is also a necessity for satisfying policy or management purposes. In service area design practice, splitting a demand unit is usually not allowed.

The SAP can be defined as a contiguity-constrained generalized assignment problem. It aims to minimize the total travel distance when using the service while satisfying constraints on facility capacity and service area contiguity. The generalized assignment problem is known to be nondeterministic polynomial time hard (NP-hard). The constraints on spatial contiguity also pose great obstacles in modeling and solving the geographic problems. Consequently, various exact and heuristic methods have been proposed for solving the SAP.

2. Problem formulation

For a geographic area, let $V = \{1, 2 \dots n\}$ be a set of n areal units. Each unit i has service demand p_i. Let $S = \{s_1, s_2 \dots s_K\}$ ($S \subset V$) be a set of K service-supply units, and each unit s_k has service

capacity q_k. Let d_{ij} be the distance between units *i* and *j*. Let c_{ij} indicate whether units *i* and *j* share a border and N_i be a set of units that are adjacent to unit *i*, $N_i = \{j | c_{ij} = 1\}$.

A MILP model can be formulated by defining the SAP as a contiguity-constrained general assignment problem. The model is similar to the districting model in Kong *et al.* (2019): only the constraints on district balance are formulated differently. Let $y_{ik} \in \{0,1\}$ denote whether unit *i* is assigned to supply unit *k*; let H_k denote a non-negative integer variable indicating the service overload of unit *k*. A model of a general assignment problem can be applied to assign each demand unit to a supply unit. The contiguity of service areas can be ensured by a network flow model (Shirabe 2005; Duque *et al.* 2011). The model is formulated as follows:

Minimize $\alpha \sum_{k \in S} H_k + \sum_{i \in V} \sum_{k \in S} p_i d_{ik} y_{ik}$ (1)

Subject to: $\sum_{k \in S} y_{ik} = 1, \forall i \in V$ (2)

$$\sum_{i \in V} p_i y_{ik} \leq q_k + H_k, \forall k \in S \quad (3)$$

$$f_{ijk} \leq (n - K) y_{ik}, \forall i \in V, j \in N_i, \forall k \in S \quad (4)$$

$$f_{ijk} \leq (n - K) y_{jk}, \forall i \in V, j \in N_i, \forall k \in S \quad (5)$$

$$\sum_{j \in N_i} f_{ijk} - \sum_{j \in N_i} f_{jik} \geq y_{ik}, \forall i \in V/S, \forall k \in S \quad (6)$$

$$\sum_{j \in N_i} f_{ijk} - \sum_{j \in N_i} f_{jik} \geq K - n, \forall i \in S, \forall k \in S \quad (7)$$

$$y_{ik} = \{0,1\}, \forall i \in V, k \in S \quad (8)$$

$$f_{ijk}, H_k \geq 0, \forall i \in V, j \in N_i, k \in S \quad (9)$$

The objective function (1) is to minimize the total travel distance from service demand units to their supply units. It also penalizes service overloads by using a large coefficient α. Constraints (2) confirm that each unit *i* is assigned to only one service-supply unit. Constraints (3) are soft constraints on service capacities. Constraints (4), (5), (6), and (7) are the flow-based formulations on contiguity, which are rewritten formulations in Duque *et al.* (2011). Constraints (8) and (9) impose restrictions on decision variables.

3. An algorithm framework

3.1 Initial solutions

Two models are alternatively used to generate initial solutions. The first model is a generalized assignment problem model as follows. The model assigns each demand unit *i* to a supply unit *k*. A random coefficient ϵ_{ik} ($|\epsilon_{ik}| < 0.02$) is used in the objective function to obtain different solutions.

$$\text{Minimize } \sum_{i\in V}\sum_{k\in S}(1+\epsilon_{ik})p_i d_{ik} y_{ik} \quad (10)$$

$$\text{Subject to: } \sum_{k\in S} y_{ik} = 1, \forall i \in V \quad (11)$$

$$\sum_{i\in V} p_i y_{ik} \le q_k, \forall k \in S \quad (12)$$

$$y_{ik} = \{0,1\}, \forall i \in V, k \in S \quad (13)$$

The second model is a transportation problem model as follows:

$$\text{Minimize } \sum_{i\in V}\sum_{k\in S}(1+\epsilon_{ik})d_{ik} y_{ik} \quad (14)$$

$$\text{Subject to: } \sum_{k\in S} y_{ik} = p_i, \forall i \in V \quad (15)$$

$$\sum_{i\in V} y_{ik} \le q_k, \forall k \in S \quad (16)$$

$$y_{ik} \ge 0, \forall i, j \in V, k \in S \quad (17)$$

The solutions from the two models need to be repaired. First, some units in a solution from the transportation model may be split into two or more parts. For each split unit, the algorithm assigns it to a service area according to the largest portion of split. Second, some areas in a solution from the two models may be nonconnective.

3.2 solution representation

The individuals in a population (solutions) are represented as integer arrays. The length of an array is equal to the number of demand units. The integer values in the array indicate the indexes of the service areas. Figure 1 shows an instance with 18 demand units and a solution with four service areas. It is coded in an integer array: [0, 1, 1, 2, 2, 3, 0, 1, 1, 2, 3, 3, 0, 0, 1, 2, 3, 3].

0	1	1	2	2	3
0	1	1	2	3	3
0	0	1	2	3	3

Figure 1. A solution for a problem instance

3.3 Crossover

Three crossover operators are designed to recombine individuals: uniform, ordered1, and ordered2. Traditional genetic algorithms use four types of crossover methods on chromosomes: single-point crossover, k-point crossover, uniform crossover, and crossover for ordered lists. Owing to the contiguity constraints on areas, various overlay-based crossover methods are used in evolutionary algorithms to solve districting problems (Chou *et al*. 2007; Datta *et al*. 2008; Xiao 2008; Liu *et al*. 2016). These crossover operations belong to the methods of the ordered-list crossover and are implemented by overlaying two solution maps. Differing from the randomly assigned indexes of districts in the districting problems, the indexes of service areas in the SAP can be predefined according to supply unit indexes. Therefore, commonly used crossover methods can be used directly in evolutionary algorithms for the SAP.

The three operators are explained using two individuals, P_1 and P_2, shown in Figure 2. In the algorithm, only one offspring is generated. Its genetics are mainly inherited from P_1: only 30% of the units or areas for the two individuals participate in the crossover operation.

P_1

0	1	1	2	2	3
0	1	1	2	3	3
0	0	1	2	3	3

P_2

0	0	1	2	2	3
0	0	1	2	3	3
0	1	1	2	2	3

Figure 2. An example of two individuals

Uniform crossover. For individual P_1, the units were selected with a possibility of 0.3, yielding the selection of some units (five gray units in Figure 3). The genetic values of the selected units in individual P_1 are replaced by the values in individual P_2. The new individual, P_3 is shown in Figure 3.

P_1

0	1	1	2	2	3
0	1	1	2	3	3
0	0	1	2	3	3

P_2

0	0	1	2	2	3
0	0	1	2	3	3
0	1	1	2	2	3

P_3

0	0	1	2	2	3
0	1	1	2	3	3
0	0	1	2	3	3

Figure 3. An example of uniform crossover

Order1 crossover. For individual P_1, the service areas were selected with a possibility of 0.3. For example, if area 1 in P_1 is selected, the genetic values of the units in area 1 were replaced with the corresponding values in individual P_2, as shown in Figure 4.

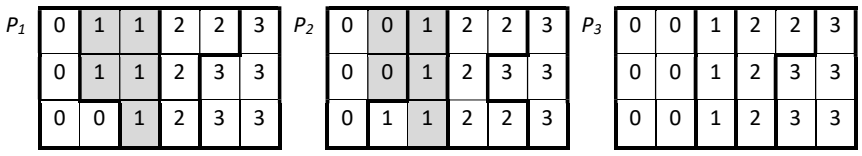


Figure 4. An example of order1 crossover

Order2 crossover. For individual P_2, the service areas were selected with a possibility of 0.3. For example, if area 1 in P_2 was selected, the genetic values of the units in area 1 were inserted into the individual P_1, as shown in Figure 5.

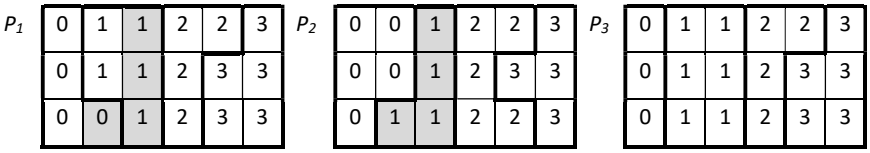
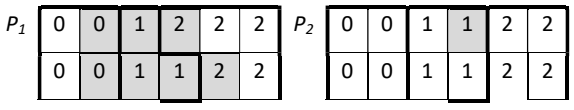


Figure 5. An example of order2 crossover

The new individual P_3 needed to be repaired, because some service areas may have been nonconnective after the crossover operation.

3.4 Mutation

The mutation operation in GAs is to maintain genetic diversity from one generation of a population to the next. There are various mutation methods in GAs. For districting problems, only the mutation on boundary units is effective; otherwise, the area continuity may be destroyed. Therefore, the mutation operator in our algorithm is as follows: (1) select the boundary units in the individual P_1 (Figure 6 left); (2) each boundary unit is possibly reassigned to its neighboring area with possibility p_m. In Figure 6 right, one unit in area 2 moves to its neighboring area 1. Note that the new solution P_2 also need to be repaired for the area connectivity.



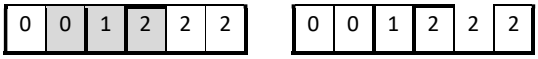
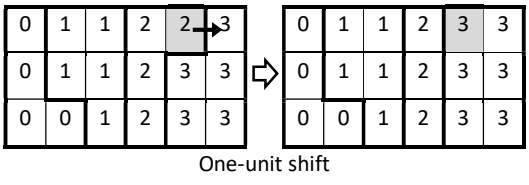


Figure 6. An example of the mutation operation

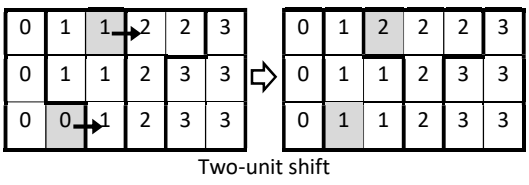
The probability of mutation has significant effects on the performance of GAs. It greatly determines the degree of solution quality and the convergence speed. Various adaptive GAs have been proposed to improve the algorithm performance; however, it is still challenging to suggest unified guidelines on parameter tuning (Elmihoub *et al*, 2006; Aleti and Moser 2016). We used a simple mechanism to adjust the probability of mutation. We divided the process of evolution into multiple periods. For example, a number of *M* iterations was considered as a period. In the first period, the initial parameter p_m was used. In the following periods, the parameter was adjusted according to the set of individuals *P''* obtained from the last period. If the difference indicator was less than 10.0%, $p_m = p_m + 0.005$; if it was greater than 20.0%, $p_m = p_m - 0.005$, otherwise, $p_m = 0.005 + \text{rand}() * 0.01$. In addition, p_m is limited between 0.005 and 0.03. For two individuals P_1 and P_2, the difference indicator was defined as the percentage of the units with different genetic values. For a set of individuals, its difference indicator was measured by the average of indicators of all individual pairs.

3.5 Local search

Three local search operators were used in the algorithm to improve the solutions generated from the crossover and mutation operations. The operators attempted to move one or more units located on the boundary to their neighboring service areas. Note that only the feasible moves were allowed, because when moving a boundary unit from its original area to a destination area, the original area may be nonconnective. Three operators—one-unit shift, two-unit shift, and three-unit shift—are illustrated in Figure 7.



One-unit shift



Two-unit shift

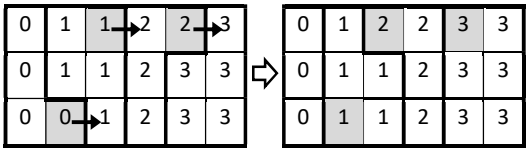


Figure 7: Examples of the local search operators

The one-unit shift operator moves boundary unit *i* to one of its neighboring area *k* as outlined in Algorithm 2. In the algorithm, all the boundary units are selected and then shuffled randomly. For each boundary unit *i*, the algorithm tries to move the boundary unit from its original area to one of its neighboring areas. The move is accepted in cases in which the original area is connective, and the new solution is better than the incumbent solution.

```
Algorithm 2: one_unit_shift(s)
(1) ulist=boundary_units(s);
(2) ulist=shuffle(ulist);
(3) s'=s;
(3) for i in ulist:
(4)   for k in neighboring_areas(i):
(5)     s'' = move(s', i, k);
(6)     if original area(i) is connective and f(s'')<f(s'):
(7)       s'=s'';
(8)       break;
(9) return s'.
```

The other two operators are similar to the one-unit shift but explore a larger neighborhood space with higher time complexity. The operator two-unit shift moves boundary unit *i* to one of its neighboring areas *k*, and at the same time moves boundary unit *j* in area *k* to one of its neighboring areas. In other words, one unit is moved into area *k*, and another unit in area *k* is moved out. Differing from the one-unit shift that involves two areas, a two-unit shift usually involves three areas. Swapping two units between two adjacent areas is a special case that involves two areas.

```
Algorithm 3: two_unit_shift(s)
(1) ulist=boundary_units(s);
(2) ulist=shuffle(ulist);
(3) s'=s;
```

```
(4) for i in ulist:
(5)   for j in ulist:
(6)     if area(i) and area(j) are not adjacent: continue;
(7)     for k in neighboring_areas(j):
(8)       s'' = move(s', i, j, k);
(9)       if original area(i) or area(j) is nonconnective: continue;
(10)      if f(s'')<f(s'):
(11)        s'=s'';
(12)        break;
(13) return s'.
```

The operator three-unit shift moves three units between, at most, four adjacent areas. It moves a boundary unit in the first area to one of its neighboring areas (the second area); then moves a boundary unit in the second area to one of its neighboring areas (the third area); then it again moves a boundary unit in the third area to one of its neighboring areas (the fourth area).

```
Algorithm 4: three_unit_shift (s)
(1) ulist=boundary_units(s);
(2) ulist =shuffle(ulist);
(3) s'=s
(4) for i in ulist:
(5)   for j in ulist:
(6)     if area(i) and area(j) are not adjacent: continue;
(7)     for h in ulist:
(8)       if area(j) and area(h) are not adjacent: continue;
(9)       for k in neighboring_areas(h):
(10)        s'' = move(s', i, j, h, k);
(11)        if original area(i), area(j) or area(h) is nonconnective: continue;
(12)        if f(s'')<f(s'):
(13)          s'= s'';
(14)          break;
(15) return s'.
```

The three operators have remarkably different computational complexity: $O(nK)$, $O(n^2K)$, and $O(n^3K)$. However, the number of possible moves is generally much fewer, because only the boundary units can be moved to their neighboring areas, and there are only one or a few

neighboring areas available for a boundary unit. The three-unit shift operator is rather expensive, but it may be highly effective to explore the neighborhood solutions in some instances.

3.6 Update population

In Algorithm 1, the population is updated immediately at the end of each loop of evolutionary iterations. To maintain population diversity, only elitist and dissimilar individuals are selected to update the population. A difference indicator the same as the definition in Section 2.4 is used to measure the dissimilarity between individuals in population *P*. The procedure for population updating is shown in Algorithm 5. In some cases, the size of the new population may be smaller than the size of *M*. However, the following process of evolution, especially the mutation operation, could generate the new individuals needed to fulfill the population size.

Algorithm 5: updatePopulation(*P*, *P′*, *P″*)
(1) Append individuals *P′* and *P′* to population *P*;
(2) Sort *P* by the objective of individual;
(3) P_{new}=*null*
(4) for *i* in *P*:
(5) *di*=100.0;
(6) for *j* in P_{new}:
(7) *di*=min(*di*, difference_indicator(*i*,*j*));
(8) if *di*>5.0:
(9) Append solution *i* to P_{new};
(10) return P_{new}.

3.7 Set partitioning

Let set Ω denote the historical service areas recorded in the evolutionary process. Each area *i* has an objective c_i and a set of units U_i in the area. An SPP model is used to select the subset service areas of Ω, which gives a minimal objective and covers all the units in set *V*. The problem is formulated as follows. The decision variable x_i indicates whether the candidate area *i* is selected. The SPP is known to be NP-hard; however, existing MIP optimizers could effectively solve the SPP instances with a considerable Ω size and problem size *V* (e.g. $|\Omega| = 20000$ and $|V| = 1000$).

Minimize $\sum_{i\in\Omega} c_i x_i$ (18)

Subject to: $\sum_{i\in\Omega, j\in U_i} x_i = 1, \forall j \in V$ (19)

$$x_i \in \{0,1\}, \forall i \in \Omega \quad (20)$$

3.8 Other algorithms

To keep the contiguity of service areas in a solution, several algorithms are frequently used in crossover, mutation, and local search operations. The first algorithm is to check the connectivity of a service area. If a spanning tree can be established using all the units in an area, the area is connective. Another approach is to use a region-growth method. Starting from a seed unit, if all the units can be assigned to their neighbors, the area is connective. The connectivity check is shown in Python-like code as follows.

```
Algorithm 6: check_connectivity(ulist)
(1) ulist1=ulist[:]
(2) ulist2=[]
(3) ulist2.append(ulist1.pop())
(4) for x in ulist2:
(5)   for i in range(len(ulist1)):
(6)     if ulist1[i] in N_x:
(7)       ulist2.append(ulist1[i])
(8)       ulist1[i]=-1
(9)   ulist1=[x for x in ulist1 if x>=0]
(10) if len(ulist1)==0: return True
(11) return False
```

The second algorithm is to find the fragmented units in a solution. We also use the region-growth method to identify the fragmented units. For each service area, a region is gradually constructed from the service-supply unit. If some units cannot be assigned to the region, they are fragmented units.

The third algorithm is to repair a solution that has fragmented units. The fragmented service areas are repaired by deleting the fragmented units and then reassigning them to their neighboring areas greedily.

Algorithm 7: repair_solution(*s*)

(1) Find the fragmented units *funits* in solution *s*;

(2) Delete *funits* from solution *s*;

(3) while *funits* is not *null*:

(4) Find all feasible assignments of units in *funits*;

(5) If there is no feasible assignment: break

(6) Select the best or second-best assignment randomly;

(7) Assign the selected unit to its neighboring area;

(8) Delete the assigned unit from *funits*;

(9) return *s*.

3.9 Implementation

The metaheuristic framework was implemented by using the Python programming language. The Python script were run in PyPy 6.0, a fast and compliant implementation of the Python language (see http://pypy.org). The generalized assignment model, the transportation model, and the SPP model were solved by the IBM ILOG CPLEX Optimizer 12.6.3.

The SAP could be solved by exact methods, local-search-based metaheuristics, population-based metaheuristics or hybrid methods. The metaheuristic framework was designed to implemented various algorithms for the SAP.

3. Algorithms

3.1 Hybrid GA with LS and SPP

The hybrid algorithm is outlined as follows.

Algorithm: Hybrid GA with LS and SPP

Parameters: population size (M), crossover possibility (p_c), mutation possibility (p_m), local search operators (*lsops*), and time limit (T).

(1) Generate population P of size M (Section 3.1);

(2) *Pool*=*null*;

(3) P_{best}=best(P);

(4) Repeat the following steps until the termination conditions are met:

(5) Select individuals P_1 and P_2 from P randomly;

(6) P'=crossover(P_1, P_2, p_c);

(7) P'=localsearch(P', *lsops*);

(8) P''=mutation(P', p_m);

(9) P''=localsearch(P'', *lsops*);

(10) P_{best}=best(P_{best}, P', P'');

(11) *Pool* =updateServiceAreaPool(*Pool*, P', P'');

(12) P =updatePopulation(P, P', P'');

(13) P'''=sppmodelling(*Pool*);

(14) Output the best solution: best(P_{best}, P''').

3.2 Population-based Iterative Local Search (ILS)

The algorithm maintains a pool of elite solution. In each iteration of search: (1) select a solution is from the pool; (2) ruin and repair the solution; (3) improve the perturbed solution by using local search operators; (4) update the pool by the new solution.

Algorithm: Iterative Local Search

Parameters: number of initial solutions (*M*), ruin method (*rm*), local search operators (*lsops*), and time limit (*T*).

(1) Generate initial solutions P;

(2) *Pool*=*null*;

(3) p_{best}=best(P);

(4) Repeat the following steps until the termination conditions are met:

(5) Select a solution p from P randomly;

(6) p'=Ruin_and_repair(*p,rm*);

(7) p''=localsearch(p', *lsops*);

(8) p_{best}=best(P_{best}, p'');

(9) P =updatePopulation(P, P', P'');

(10) *Pool* =updateServiceAreaPool(*Pool*, p'');

(11) p'''=sppmodelling(*Pool*);

(12) Output the best solution: best(P_{best}, p''').

3.3 Population-based Iterative Local Search with VND (ILS+VND)

Algorithm: ILS+VND

Parameters: number of initial solutions (*M*), ruin method (*rm*), local search operators (*lsops*), and time limit (*T*).

(1) Generate initial solutions *P*;

(2) *Pool*=*null*;

(3) p_{best}=best(*P*);

(4) Repeat the following steps until the termination conditions are met:

(5) Select a solution *p* from *P* randomly;

(6) *p'*=Ruin_and_repair(*p,rm*);

(7) *p''*=VNDsearch(*p'*, *lsops*);

(8) p_{best}=best(P_{best}, *p''*);

(9) *Pool* =updateServiceAreaPool(*Pool*, *p''*);

(10) *P* =updatePopulation(*P*, *p''*);

(11) *p'''*=sppmodelling(*Pool*);

(12) Output the best solution: best(P_{best}, *p'''*).

3.4 Population-based Variable Neighborhood Search (VNS)

Algorithm: VNS

Parameters: number of initial solutions (*M*), ruin method (*rm*), local search operators (*lsops*), and time limit (*T*).

(1) Generate initial solutions *P*;

(2) *Pool*=*null*;

(3) p_{best}=best(*P*);

(4) Repeat the following steps until the termination conditions are met:

(5) Select a solution *p* from *P* randomly;

(6) *k=1*

(7) *while k>K:*

(8) *p'=shake(p, k)*

(9) *p''*=VNDsearch(*p'*, *lsops*);

(10) if *p'' is better that p: p= p'', k=1*

(11) else: k=k+1

(12) p_{best}=best(P_{best}, *p''*);

(13) *Pool* =updateServiceAreaPool(*Pool*, *p''*);

(14) *P* =updatePopulation(*P*, *p''*);

(15) *p'''*=sppmodelling(*Pool*);

(16) Output the best solution: best(P_{best}, p''').

3.5 Multi-start simulated annealing (SA)

Algorithm: SA

Parameters: number of starts (*M*), initial temperature (t0), number of iterations (*loops*), local search operators (*lsops*), and time limit (*T*).

(1) *Pool*=*null*;

(2) p_{best}=null;

(3) coolingrate= math.exp(math.log(0.005)/loops);

(4) Repeat the following steps until the termination conditions are met:

(5) Generate an initial solution *p*;

(6) for loop in range (loops):

(7) *t=t0** pow*(coolingrate, loop);*

(8) *p*, p_{best} = localsearch(*p*, *lsops, t*);

(9) if f(*p*)<f(p_{best}): p_{best} =*p*

(10) *Pool*=updateServiceAreaPool(*Pool*, *p*);

(11) *p'*=sppmodelling(*Pool*);

(12) Output the best solution: best(P_{best}, *p'*).

3.6 Multi-start hill climbing

Algorithm: multi-start hill climbing

Parameters: local search operators (*lsops*), search time limit (*T*).

(1) *Pool*=*null*;

(2) p_{best}=null;

(3) Repeat the following steps until the termination conditions are met:

(4) Generate an initial solution *p*;

(5) Repeat the following steps until the solution *p* cannot be improved:

(6) *p* = localsearch(*p*, *lsops*);

(7) if f(p)<f(p_{best}): p_{best}=p;

(8) *Pool* =updateServiceAreaPool(*Pool*, *p*);

(9) *p*=sppmodelling(*Pool*);

(10) Output the best solution: best(P_{best}, *p*).

3.7 Multi-start variable neighborhood decent (VND)

Algorithm: multi-start VND

Parameters: local search operators (*lsops*), search time limit (*T*).

(1) *Pool*=*null*;

(2) p_{best}=null;

(3) Repeat the following steps until the termination conditions are met:

(4) Generate an initial solution *p*;

(5) *p* = vnd_localsearch(*p*, *lsops*);

(6) if f(p)<f(p_{best}): p_{best}=p;

(7) *Pool* =updateServiceAreaPool(*Pool*, *p*);

(8) *p*=sppmodelling(*Pool*);

(9) Output the best solution: best(P_{best}, *p*).

3.8 Multi-start record-to-record travel search (RRT)

Algorithm: multi-start RRT

Parameters: deviation (*dev*), local search operators (*lsops*), search time limit (*T*).

(1) *Pool*=*null*;

(2) p_{best}=null;

(3) Repeat the following steps until the termination conditions are met:

(4) Generate an initial solution *p*;

(5) p_{cbest}=*p* #current best solution

(6) not_improved=0

(7) Repeat the following steps until not_improved>10:

(8) *p*, p_{cbest} = localsearch(*p*, *lsops, dev*);

(9) if f(p_{cbest})<f(p_{best}): p_{best}= p_{cbest};

(10) *Pool* =updateServiceAreaPool(*Pool*, *p*);

(11) not_improved+=1

(12) if p_{cbest} is updated: not_improved=0

(13) *p*=sppmodelling(*Pool*);

(14) Output the best solution: best(P_{best}, *p*).

3.9 Multi-start simple stochastic local search (SLS)

Algorithm: multi-start SLS

Parameters: local search operators (*lsops*), search time limit (*T*), possibility of moving to a random neighbor (*r*)

(1) *Pool*=*null*;

(2) p_{best}=null;

(3) Repeat the following steps until the time limit is reached:

(4) Generate an initial solution *p*;

(5) p_{cbest}=*p* #current best solution

(6) not_improved=0

(7) Repeat the following steps until not_improved>10:

(8) *p* = localsearch(*p*, *lsops, r*);

(9) *p,* p_{cbest} = localsearch(*p*, *lsops*);

(10) if f(p_{cbest})<f(p_{best}): p_{best}= p_{cbest};

(11) *Pool* =updateServiceAreaPool(*Pool*, *p*);

(12) not_improved+=1

(13) if p_{cbest} is updated: not_improved=0

(14) *p*=sppmodelling(*Pool*);

(15) Output the best solution: best(P_{best}, *p*).

3.10 commonly used algorithms

Algorithm: localsearch(*s*, *ops, rule*)

Parameters: current solution *s*, local search operators (*ops*)

(1) for *op* in *ops*:

(2) *S* = neightborhood_solutions(*s*, *op*)

(3) for *s'* in *S*:

(4) if solution s' is non-connective: continue

(5) if f(*s'*) < f(*s*): *s*=*s'*

(6) else if the acceptance rule can be satisfied: *s*=*s'* # acceptance rules such as simulated annealing, deviation threshold, or stochastic acceptance

(7) return *s*

4. Problem instances

Three study areas, ZY, GY, and GY2 in Kong *et al*. (2019), were adapted for delineating the service areas. There are 324, 297 and 1276 areal units in the study areas, respectively. Figure 8 shows all the areal units and the service demand in each areal unit. Some units in the study areas were assumed to be service-supply units, and were used to generate problem instances.

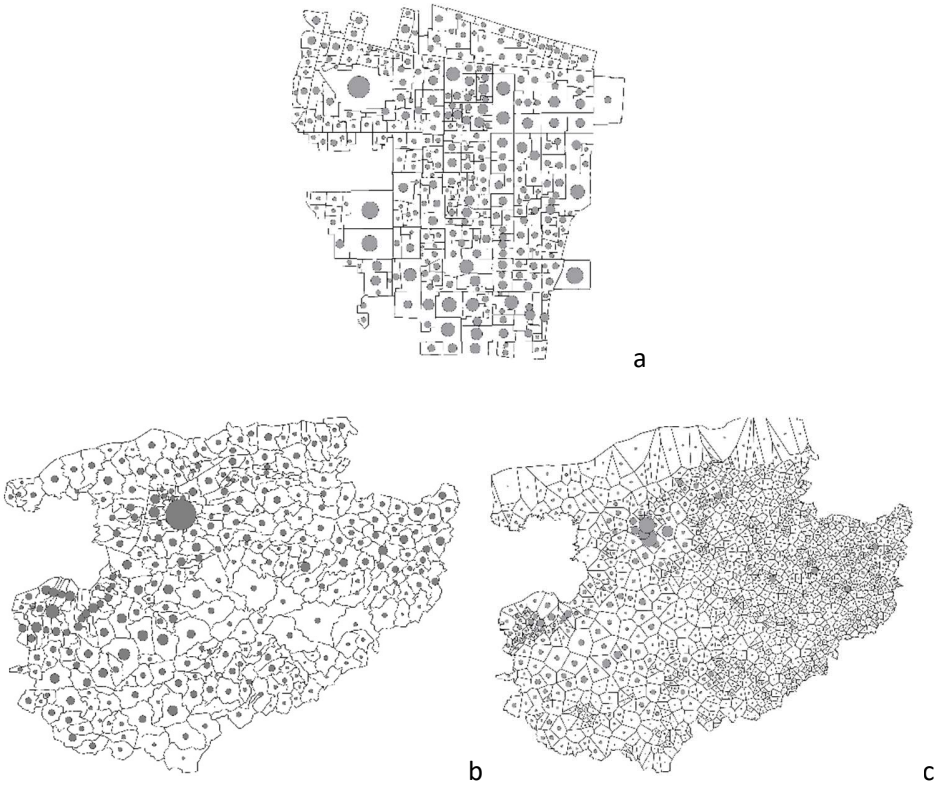


Figure 8: The areal units in areas ZY (a), GY (b) and GY2 (c) (Kong *et al*. 2019). The grey circles indicate the size of the service demand in each areal unit.

Using the study areas, 12 sets of instances were prepared to test the hybrid algorithm. For each study area, four types of facility configurations were designed as follows. (1) Select a number of facilities from candidates randomly (Type A); (2) Select a number of facilities from candidates by solving a location-allocation model (Type B); (3) Adjust the facility capacities in each instance of Type A and Type B to ensure that the supply-demand ratio is around 1.03 (denoted as Type C and Type D, respectively). Finally, 60 instances were prepared. Table 1 shows the supply-demand ratios of the 12 sets of instances.

Table 1. Supply-demand ratios of 12 sets of instances

Area	No. facilities	Type A	Type B	Type C	Type D
ZY	13-17	1.056-1.161	1.092-1.237	1.025-1.029	1.024-1.033
GY	37-41	1.060-1.110	1.065-1.114	1.025-1.034	1.026-1.034
GY2	18-22	1.007-1.190	1.069-1.213	1.027-1.034	1.026-1.033

5. Solution results from the hybrid algorithm (GA+LS+SSP)

The hybrid algorithm described in Section 3.1 was used to solve the problem instances. The Python script were run in PyPy 6.0, a fast and compliant implementation of the Python language (see http://pypy.org). The generalized assignment model, the transportation model, and the SPP model were solved by the IBM ILOG CPLEX Optimizer 12.6.3. Each instance was repeatedly solved for ten times. To verify the optimality of the solutions, the instances were also solved by the MILP model formulated in Section 2. The algorithm ran on a desktop computer with Intel Core I7-6700 CPU 3.40-GHz, 8-GB RAM and the Windows 10 operating system.

In the experiment, the population size (*M*) was set to 20, and the crossover possibility (P_c) was 0.7. For the crossover operations, the three methods in Section 3.3 were selected randomly. The initial mutation possibility (P_m) was set to 0.03 and was periodically adjusted according to the historical search information. Three operators (one-unit shift, two-unit shift, and three-unit shift) were used in the local search. The coefficient α in objective function (1) was set to 10000. This is large enough to penalize the service overloads in service areas. The iteration time (*T*) was limited to 300 seconds. The evolutionary process also terminated if the relative gap between the incumbent objective and the lower bound was less than 0.1%. The lower bound for the problem instance was estimated by the transportation model with coefficient $|\epsilon_{ik}| = 0$. To analyze the sensitivity of the parameter settings, the instances were also solved using other parameters, such as *M*=10, P_c=0.0, *T*=150, and fewer local search operators.

Additionally, two parameters, *MipGap* and *Timelimit,* were set for the CPLEX optimizer: *MipGap* =0.001 for solving the transportation model, *MipGap* =0.003 for the assignment model, *MIPGap*=0.001 and *Timelimit*=300 for the SPP model, and *MIPGap*=10^{-10} and *Timelimit*=7200 for the MILP model described in Section 2.

The solution results from the 60 instances are listed in Table 2. The instance name consists of the area name, the number of facilities and the type of facility configurations. For each instance, the solution objective, optimality gap, and computation time obtained from the CPLEX optimizer are shown in columns *Obj*, *MIPGap,* and *Time*. Among the model solutions, 50 are optimal and 10 are near-optimal with *MIPGap* between 0.00%–0.32%. For each instance, there are 10 solutions obtained from the hybrid algorithm. The average and best objectives, the relative deviation, and the average computation time in seconds are shown in columns *Obj*, *Best*, *Dev,* and *Time,* respectively. In addition, the solution gaps to the optimal or near-optimal model solutions are shown in column *Gap*. In column *Best*, the optimal values are shown with underlines, and the best objective from instance gy41c is better than the model solution.

Table 2. Solution results from 60 instances

Instance	MILP Model			Hybrid algorithm				
	Obj	MIPGap	Time	Obj	Dev	Best	Time	Gap
zy13a	2755.73	opt	365.89	2757.87	0.05%	2755.88	314.05	0.08%
zy14a	2505.25	opt	144.47	2506.73	0.07%	2505.45	316.82	0.06%
zy15a	2196.25	opt	91.69	2198.44	0.05%	2197.03	320.94	0.10%
zy16a	1995.61	opt	54.44	1997.88	0.05%	1995.61	322.41	0.11%
zy17a	1972.27	opt	312.98	1973.14	0.04%	1972.27	321.24	0.04%
gy37a	76856.92	opt	1000.99	76934.14	0.10%	76857.83	336.12	0.10%
gy38a	75941.88	opt	1170.77	75977.01	0.04%	75941.88	337.78	0.05%
gy39a	75193.32	opt	1234.56	75247.23	0.10%	75193.32	340.24	0.07%
gy40a	74046.95	opt	170.88	74075.34	0.10%	74046.95	333.15	0.04%
gy41a	73368.85	opt	739.48	73376.37	0.02%	73368.85	331.31	0.01%
gy218a	2377323.84	0.07%	7201.48	2379043.61	0.02%	2378393.27	419.43	0.07%
gy219a	2123389.97	0.02%	7204.95	2126186.41	0.02%	2125429.15	438.19	0.13%
gy220a	2012878.78	0.01%	7206.06	2013648.68	0.01%	2013450.48	398.16	0.04%
gy221a	1931624.28	opt	84.78	1931708.91	0.00%	1931624.57	308.54	0.00%
gy222a	1875355.92	opt	47.84	1875449.78	0.01%	1875355.92	83.37	0.01%
zy13b	1897.42	opt	91.39	1897.46	0.01%	1897.42	316.87	0.00%
zy14b	1755.88	opt	149.28	1758.84	0.06%	1756.12	313.67	0.17%
zy15b	1656.53	opt	78.72	1657.08	0.09%	1656.53	314.73	0.03%
zy16b	1591.86	opt	121.04	1594.48	0.21%	1591.86	313.94	0.16%
zy17b	1536.53	opt	73.32	1539.85	0.22%	1536.53	314.57	0.22%
gy37b	75105.07	opt	1137.55	75189.05	0.14%	75105.07	332.04	0.11%
gy38b	73943.37	opt	527.36	74064.14	0.15%	73943.37	330.82	0.16%
gy39b	72908.70	opt	188.30	73059.92	0.23%	72908.70	332.26	0.21%
gy40b	72166.58	opt	147.28	72261.15	0.16%	72166.58	333.06	0.13%
gy41b	71202.99	opt	252.14	71333.72	0.28%	71202.99	331.39	0.18%

gy218b	1986777.50	opt	159.29	1987294.75	0.00%	1987205.13	73.18	0.03%
gy219b	1927032.00	opt	67.03	1927080.57	0.00%	1927056.38	71.89	0.00%
gy220b	1904637.49	opt	74.10	1904691.44	0.00%	1904655.61	73.43	0.00%
gy221b	1889743.21	opt	86.73	1889803.24	0.00%	1889785.11	76.34	0.00%
gy222b	1875200.30	opt	59.46	1875253.13	0.00%	1875232.58	77.61	0.00%
zy13c	2822.23	opt	43.70	2824.52	0.09%	2823.02	313.53	0.08%
zy14c	2589.93	opt	514.55	2595.40	0.09%	2590.40	315.50	0.21%
zy15c	2405.40	opt	399.81	2406.58	0.07%	2405.40	320.49	0.05%
zy16c	2260.01	opt	274.39	2261.73	0.04%	2260.67	320.01	0.08%
zy17c	2250.94	opt	575.95	2254.50	0.10%	2250.94	322.98	0.16%
gy37c	79924.28	0.12%	7204.52	80109.31	0.16%	79976.92	386.22	0.23%
gy38c	79514.67	opt	1112.73	79719.15	0.28%	79514.67	368.15	0.26%
gy39c	78748.95	opt	1280.30	78885.67	0.20%	78748.95	358.76	0.17%
gy40c	78402.60	opt	1088.69	78530.74	0.16%	78402.60	365.35	0.16%
gy41c	79771.00	0.21%	7203.73	79815.19	0.05%	**79761.12***	380.13	0.06%
gy218c	2317477.99	0.03%	7204.80	2318634.58	0.02%	2318183.92	413.01	0.05%
gy219c	2142243.12	0.03%	7205.44	2144887.10	0.02%	2143896.10	419.01	0.12%
gy220c	2119263.56	0.04%	7205.84	2120664.49	0.02%	2119935.15	410.43	0.07%
gy221c	2026852.86	0.00%	7205.17	2027121.62	0.00%	2027059.69	88.77	0.01%
gy222c	2003951.77	0.01%	7204.17	2004422.04	0.02%	2004010.93	396.66	0.02%
zy13d	2050.88	opt	77.56	2051.57	0.04%	2050.88	318.58	0.03%
zy14d	1927.72	opt	202.80	1930.32	0.03%	1929.59	320.32	0.13%
zy15d	1874.58	opt	104.13	1875.61	0.07%	1874.58	319.55	0.06%
zy16d	1840.85	opt	787.34	1843.48	0.13%	1840.85	319.62	0.14%
zy17d	1841.48	opt	3965.80	1843.05	0.04%	1842.56	347.76	0.09%
gy37d	76744.96	opt	1090.05	76790.45	0.18%	76744.96	325.88	0.06%
gy38d	77792.10	opt	2035.78	77839.09	0.10%	77792.10	383.80	0.06%
gy39d	79150.72	0.24%	7206.91	79288.95	0.19%	79150.72	359.62	0.17%
gy40d	78572.69	0.32%	7205.88	78706.86	0.14%	78572.69	380.82	0.17%
gy41d	80533.51	0.13%	7205.42	80563.55	0.05%	80533.51	335.40	0.04%
gy218d	2016743.88	opt	6941.55	2017919.38	0.02%	2017364.50	330.80	0.06%
gy219d	1986365.28	opt	929.58	1986594.03	0.00%	1986534.72	72.04	0.01%
gy220d	2026991.94	0.01%	7205.20	2027848.57	0.01%	2027615.56	160.34	0.04%
gy221d	2016529.84	0.02%	7208.63	2017237.65	0.04%	2016643.33	392.89	0.04%
gy222d	2028631.85	0.03%	7207.47	2029956.86	0.02%	2029574.51	409.68	0.07%

The results in Table 2 show that the hybrid algorithm is robust and effective. First, the objective deviations are small, ranging from 0.00% to 0.28%. Second, the solutions approximate to the optimal or near-optimal model solutions with an average gap of 0.08% and ranging from

0.00% to 0.26%. Third, among the best solutions, 29 are optimal and one is better than the model solution.

6. How to use the algorithm framework

6.1 System requirements

Python recommended: Python 2.7.x on Windows, and pypy 6.0 (a fast and compliant Python, http://pypy.org).

MIP Solver: ILOG CPLEX V12.6 or above.

Python API: PuLP 1.6.0 (https://pypi.org/project/PuLP/), cplex (CPLEX Python API)

Download the files and copy all the files in a directory.

6.2 Solve the problem instances

Run Python scripts to solve a problem instance. For examples:

- mip.py (solve the problem by exact method)

```
import algorithm_framework_for_SAP as d
d.mip_solver=”cplex” # using CPLEX solver only
d.solver_mipgap=0.00000000001 # mipgap for CPLEX
d.readfile("zys_13b.txt", "zys_connectivity.txt") # read an instance: unit file and connectivity file
d.mipmodel() # create a MILP file, and sovle it by CPLEX
d.print_solution()
```

- …\pypy hybrid.py (solve the problem by the hybrid method)

```
import algorithm_framework_for_SAP as d
#d.operators_selected=[0]
# [0] - use one-unit-move operator
# [0,1] - use one-unit-move and two-unit-move operators
# [0,1,2] - use three operator, default
#d.mip_solver=”cplex”   #”cbc” (coin cbc solver), ”cplex”(ILOG CPLEX solver) or “” (no any mip solver)
d.readfile("zys_13a.txt", "zys_connectivity.txt") # read an instance: unit file and connectivity file
d.ga(13,20,300, -1, 0.7, 0.03, 1,-1) # solve the problem by hybrid heuristic GA+LS+SPP
# 13: number of facilities/service areas
# 20: population size
# 300: time limit in seconds
# -1: cross over method. 0 uniform, 1 order1, 2 order2, 9 no crossover, -1 random method
# 0.7: crossover possibility of 0.3=1-0.7
```

```
# 0.03: mutation possibility
# 1: use SSP modeling? 1 yes, 0 no.
# -1: random seed
d.print_solution() # print the final solution
```

- …\pypy heur.py (solve the problem by local search method)

```
import algorithm_framework_for_SAP as d
#d.operators_selected=[0,1,2]
#d.mip_solver="cplex"   #"cbc" (coin cbc solver), "cplex"(ILOG CPLEX solver) or "" (no any mip solver)
d.readfile("zys_13b.txt", "zys_connectivity.txt") # read an instance: unit file and connectivity file
d.set_solver_params(13,"ils",10,300,1,-1)
#solver parameters:
#13 - number of facilities/service areas
#"ils" - algorithm such as ils, ilsvnd, vns, sa, hc, vnd, rrt, sls
#10 - number of initial solutions, only for the algorithms such as ils, ilsvnd and vns
#300 – search time limit in seconds
#1 - spp modeling, 1 (spp modeling) or 0 (spp modeling)
#-1 - random seed, 0-100 or -1 (random seed generated automatically)
d.solve() #solve the instance
d.print_solution()
```

6.3 Solve your problem instances

Prepare your instance data by using GIS or any other tools. The data format for the files are shown as follows.

(1) Attributes of areal units

ID	Demands	X	Y	FCadidature	FCost	FCapacity
1	8	64201.83203	43952.77734	0	110	200
2	5	64600.07031	43983.01953	0	0	0
……						

The attributes of each areal unit are listed in one text line.

- ID: the unique ID of the areal unit
- Demands: the service demands in the areal unit
- X: x coordinate
- Y: y coordinate, used to calculate geometric distance.
- FCadidature: reserved

- FCost: reserved
- FCapacity: the service capacity supplied by the areal unit

(2) Adjacent links

OID	ID1	ID2	DIST
0	1	25	0.000000
1	2	11	0.000000
……			

Each line records one adjacent link between two areal units.

- OID: the record ID
- ID1: the ID of the first unit
- ID2: the ID of the second unit
- DIST: reserved

(3) Network distances

The distance between any two units are calculated by default. If a distance file is provided, all the distances will be replaced by the new distances.

OID	ID1	ID2	DIST
0	1	25	2.459
1	2	11	1.345
……			

- OID: the record ID
- ID1: the ID of the first unit
- ID2: the ID of the second unit
- DIST: the distance between the two units

The distance file can be read by function: readdistance(dfile).