On the implementation of a swap-based local search procedure for the *p*-median problem

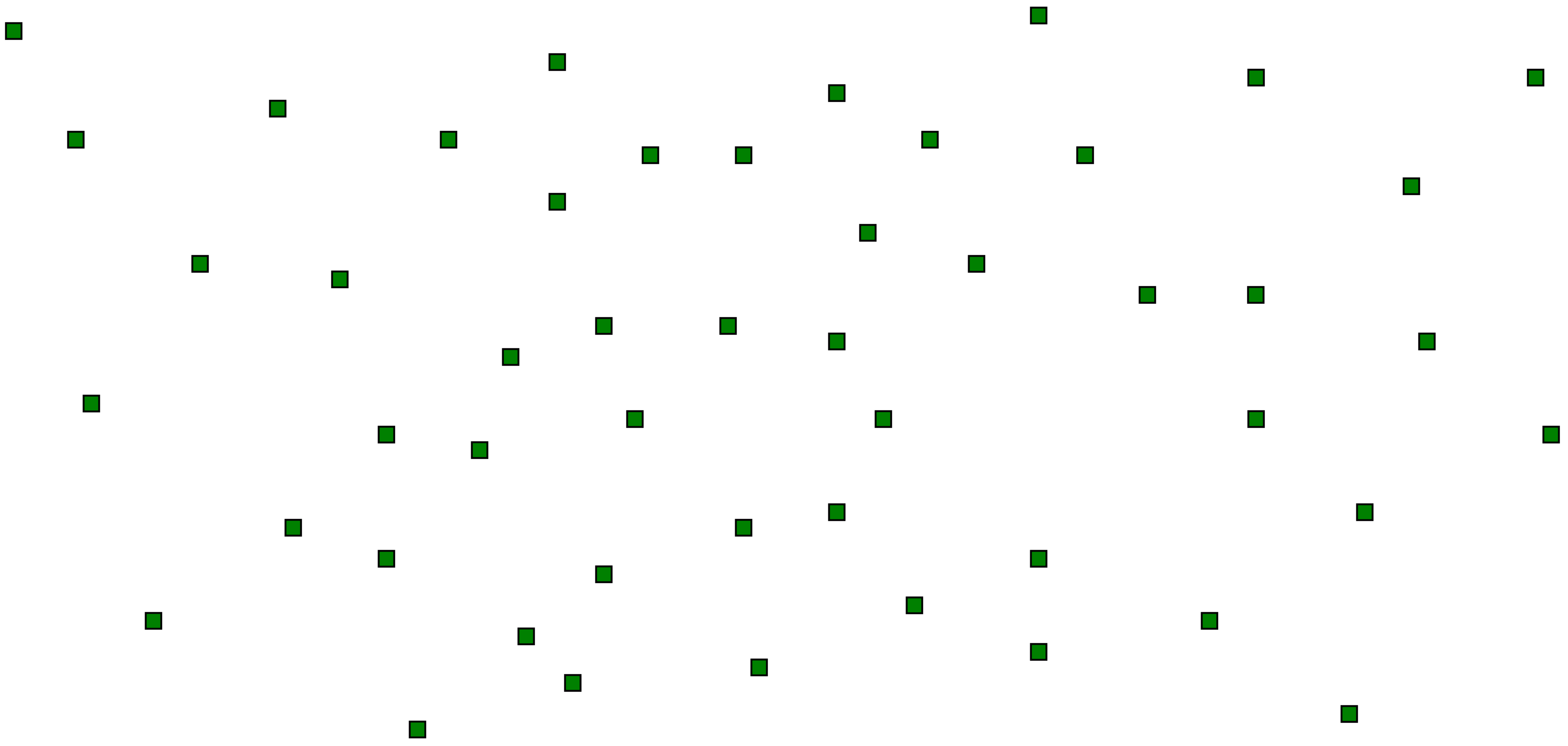
Mauricio G. C. Resende
AT&T Labs Research

Renato F. Werneck
Princeton University
(Research done while at
AT&T Labs Research)

The *p*-median Problem

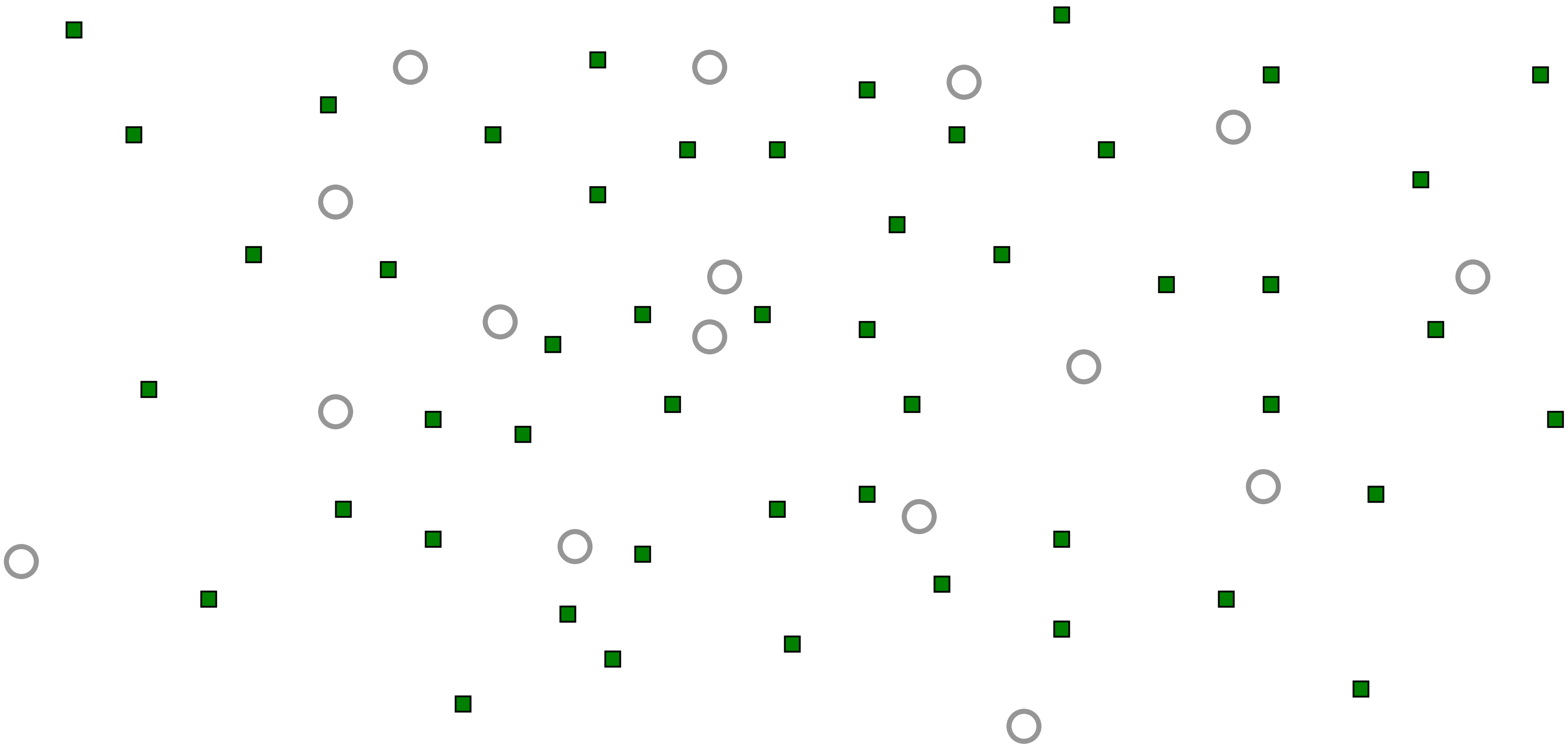
- Also known as the *k*-median problem.
- Input:
  - a set U of n *users* (or *customers*);
  - a set F of m *potential facilities;*
  - a distance function (d: $U \times F \rightarrow \Re$);
  - the number of facilities p to open ($0 < p < m$).
- Output:
  - a set $S \subseteq F$ with p open facilities.
- Goal:
  - minimize the sum of the distances from each user to the closest open facility.

Example



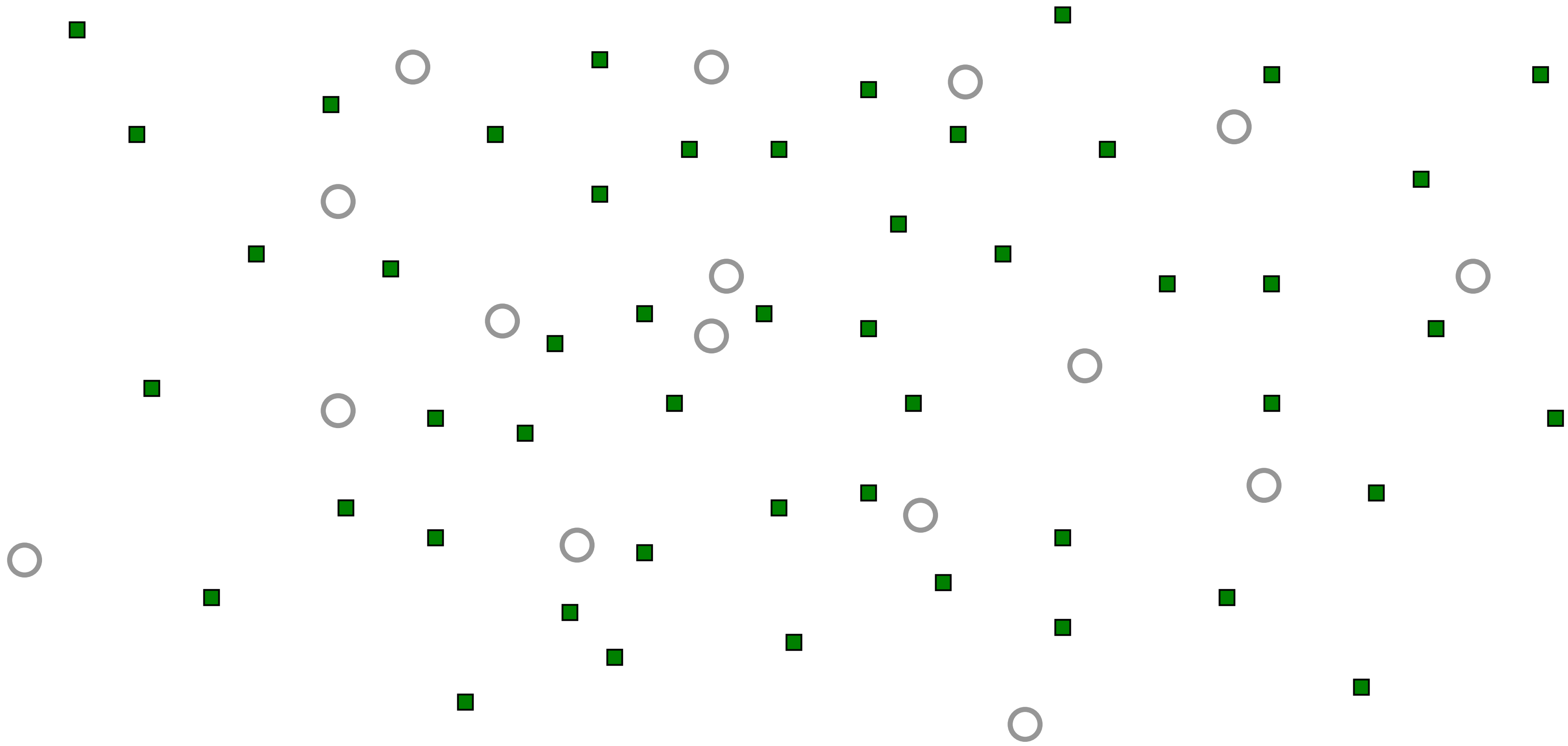
50 customers

Example



16 potential facilities

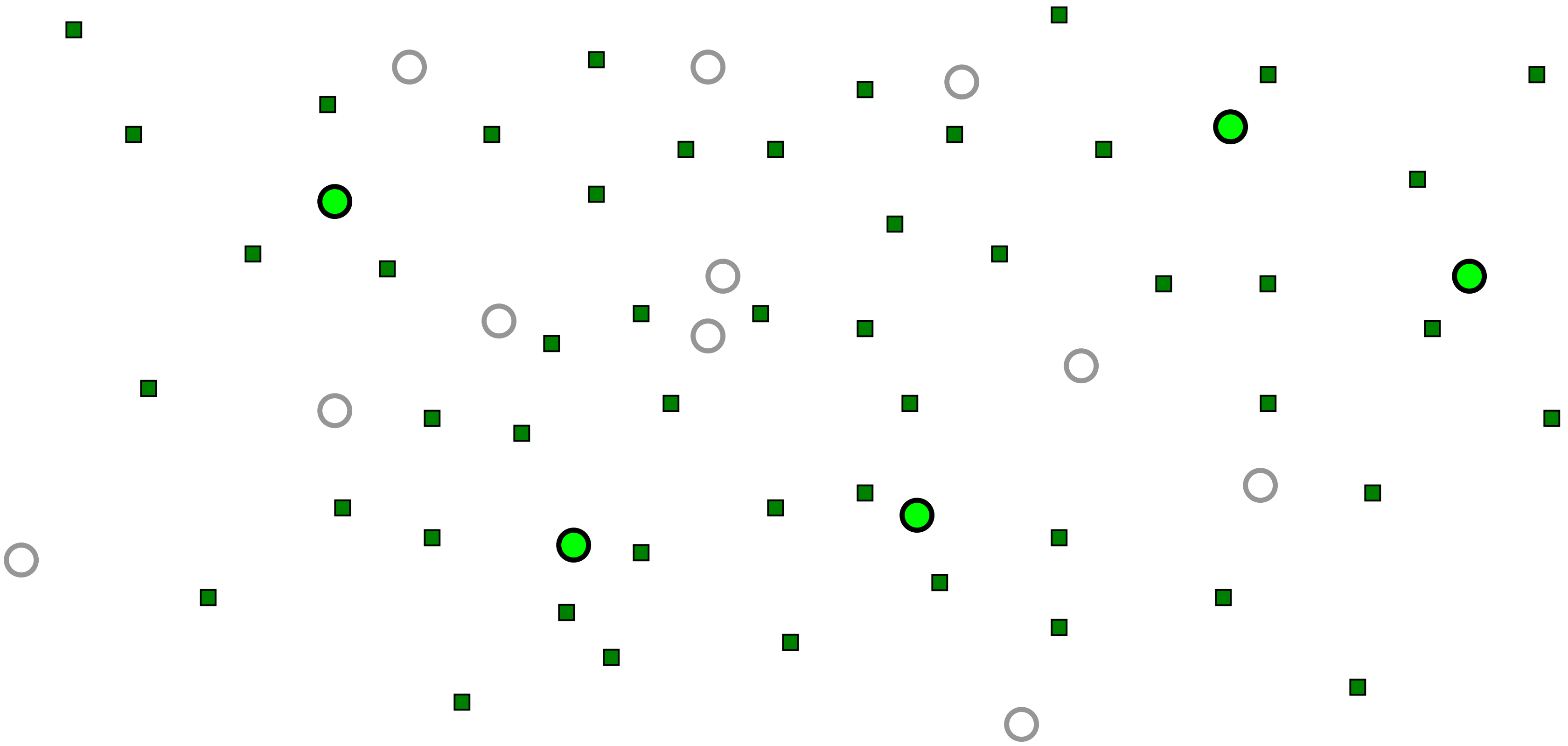
Example



assume p=5

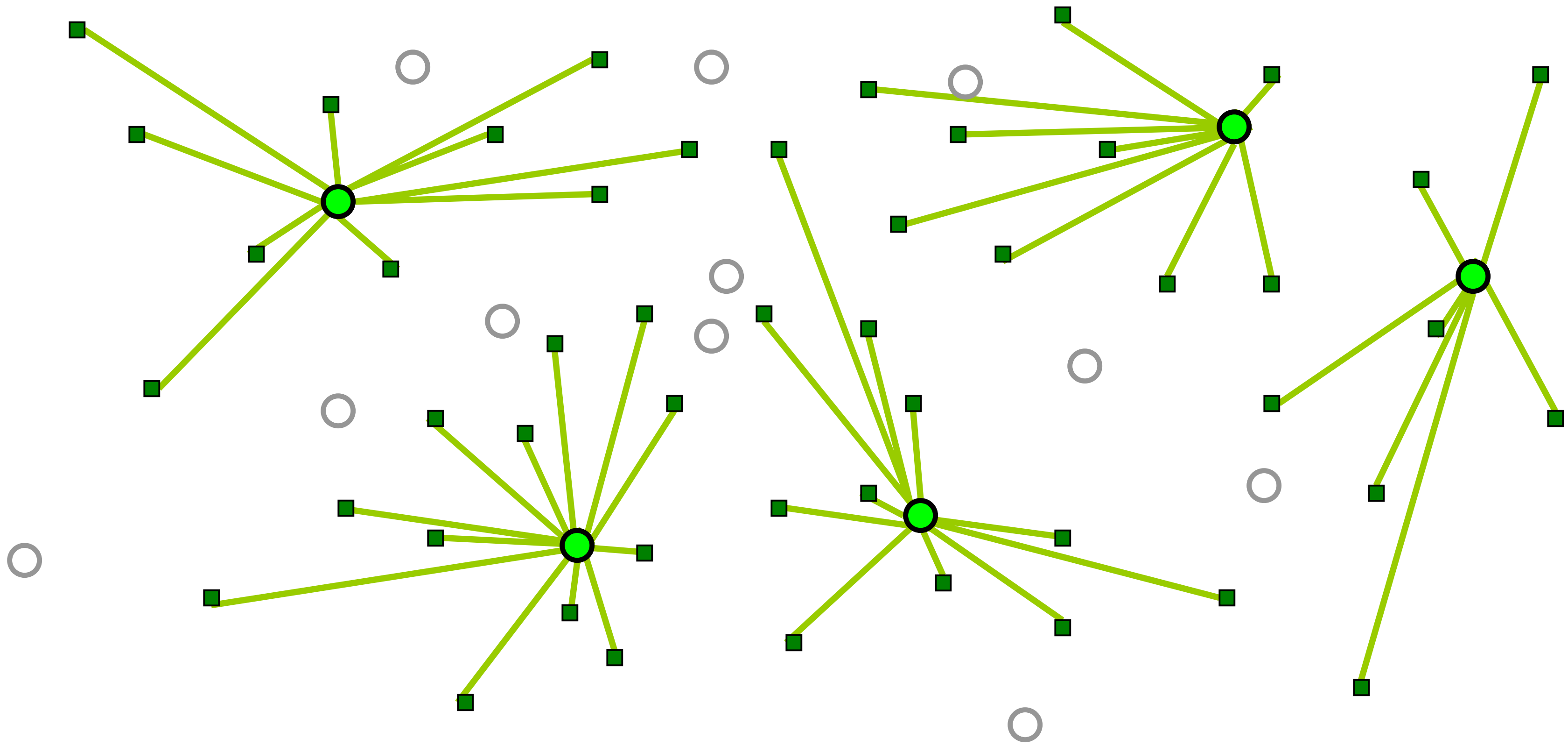
(5 facilities will be opened)

Example



This is a valid solution.

Example



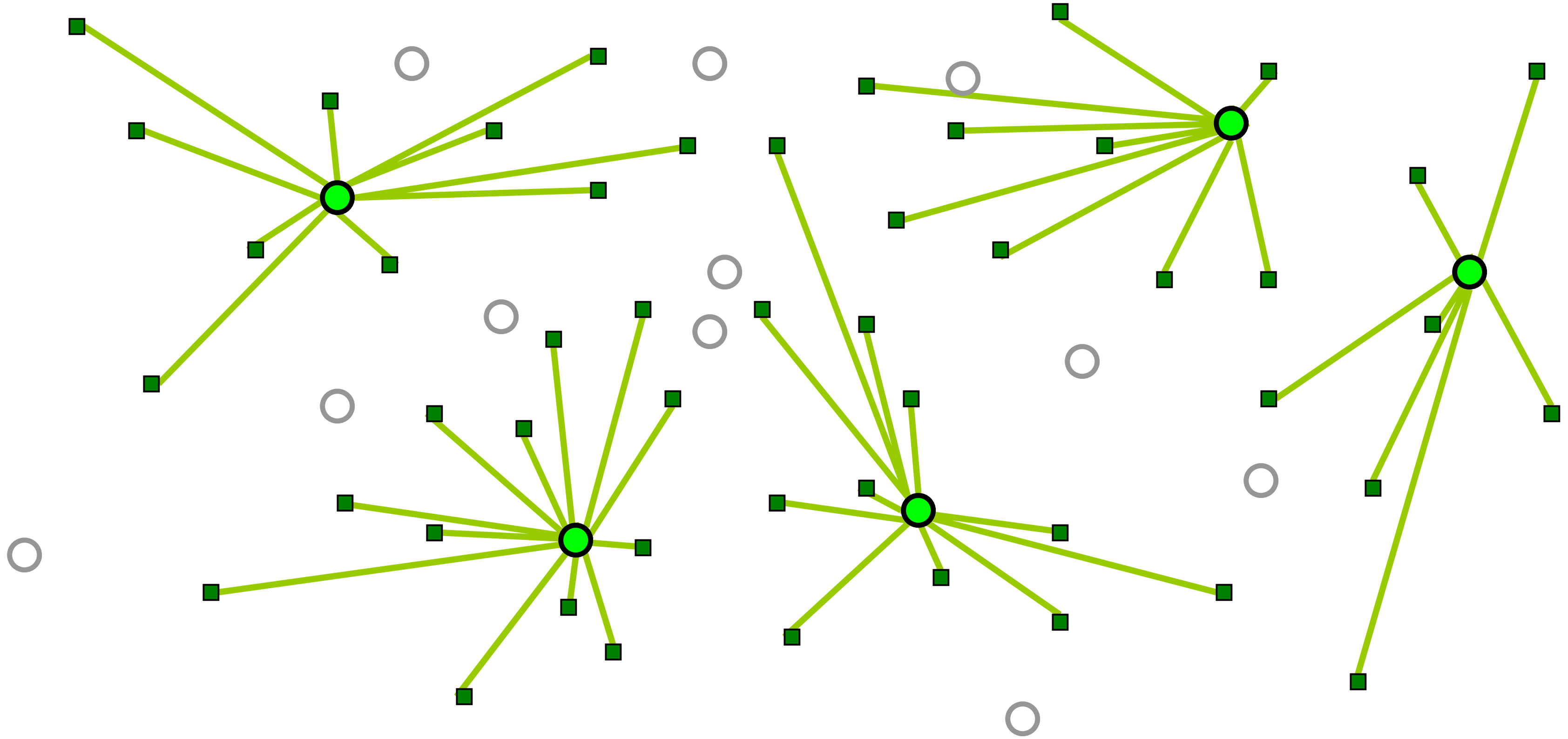
This is a valid solution with the proper assignments.

Local Search

Basic Steps:

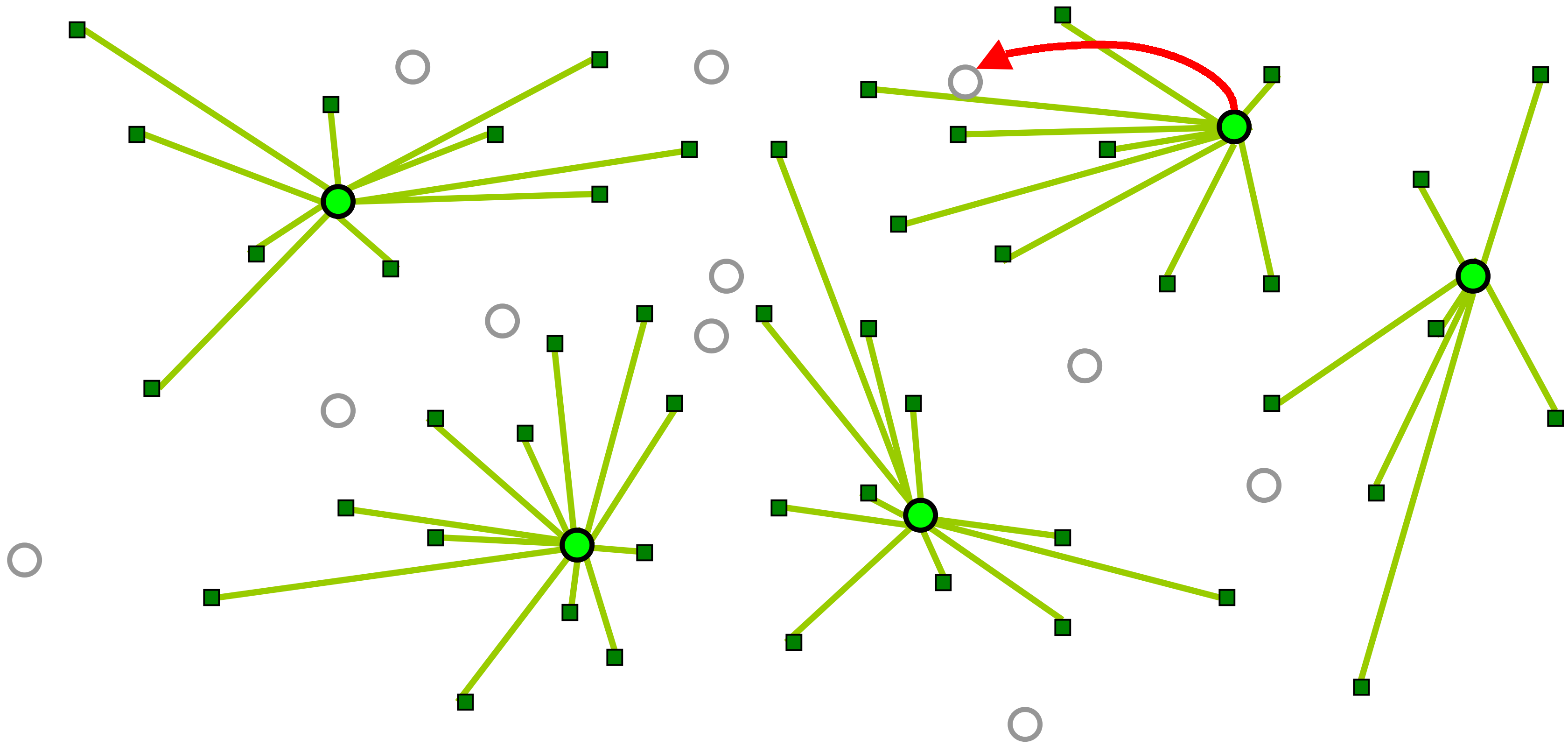
1. Start with some valid solution.
2. Look for a pair of facilities (f_i, f_r) such that:
   - f_i does **not** belong to the solution;
   - f_r **belongs** to the solution;
   - swapping *i* and *r* improves the solution.
3. If (2) is successful, swap f_i and f_r and repeat (2); else stop (a *local minimum* was found).

Local Search - Example



original solution

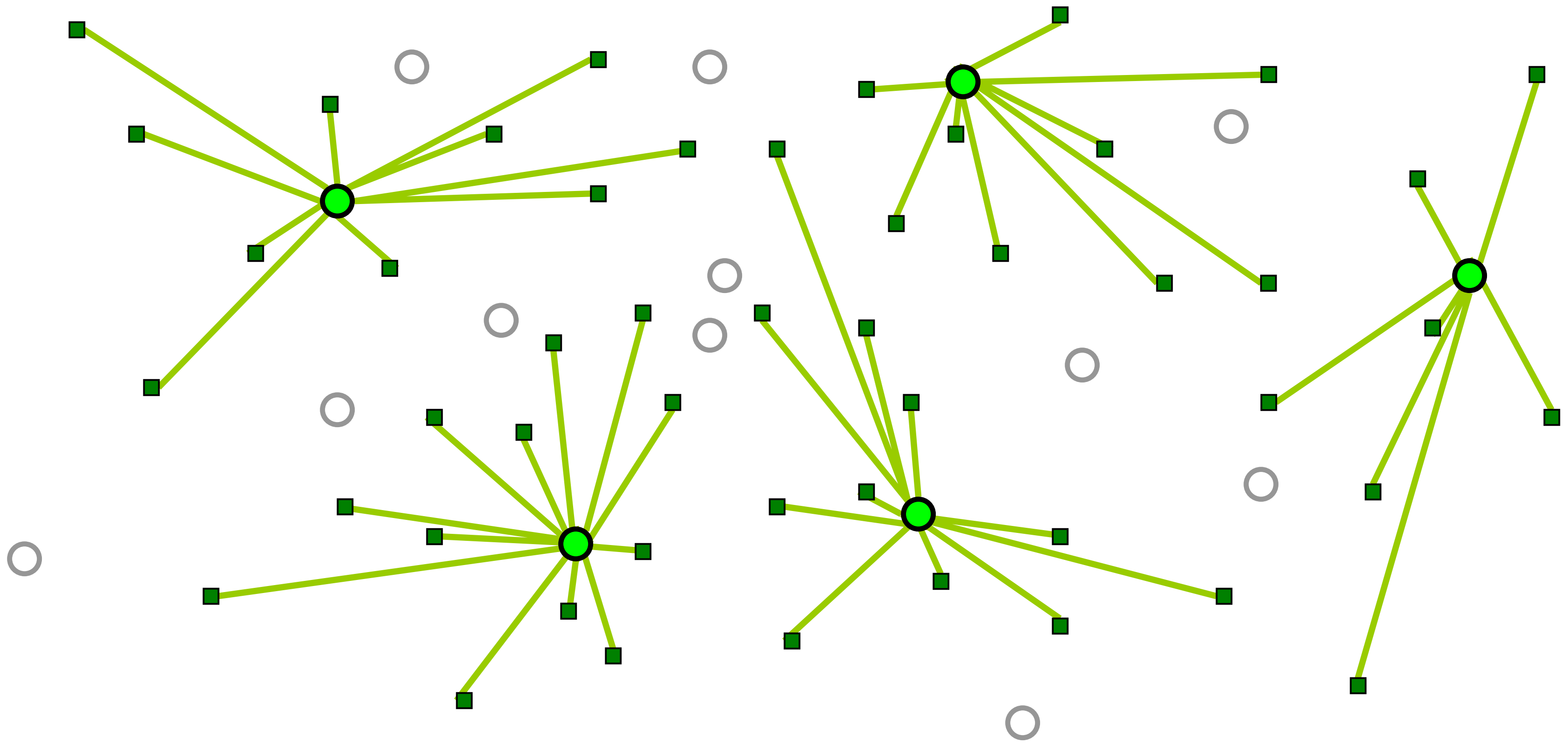
Local Search - Example



original solution

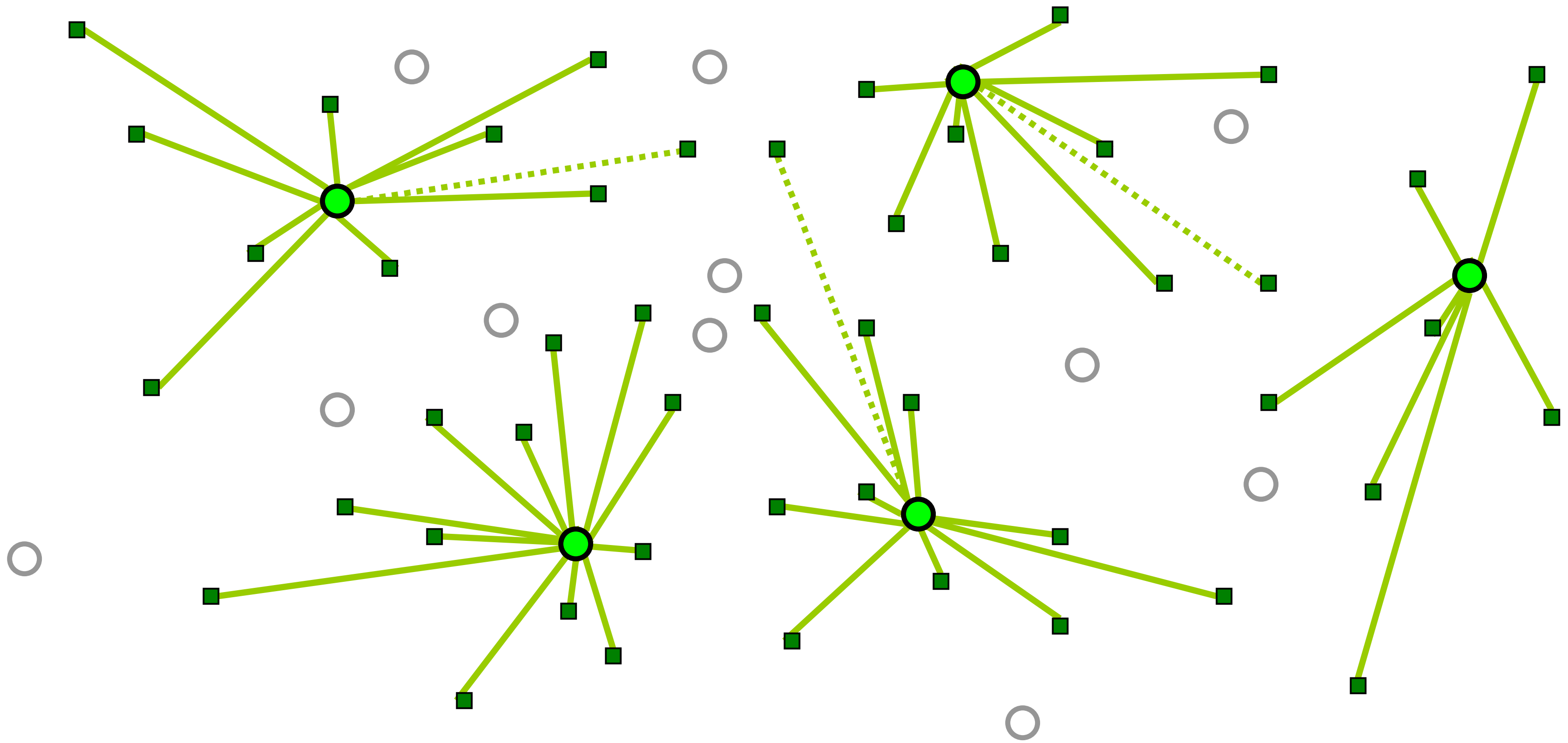
(not a local optimum)

Local Search - Example



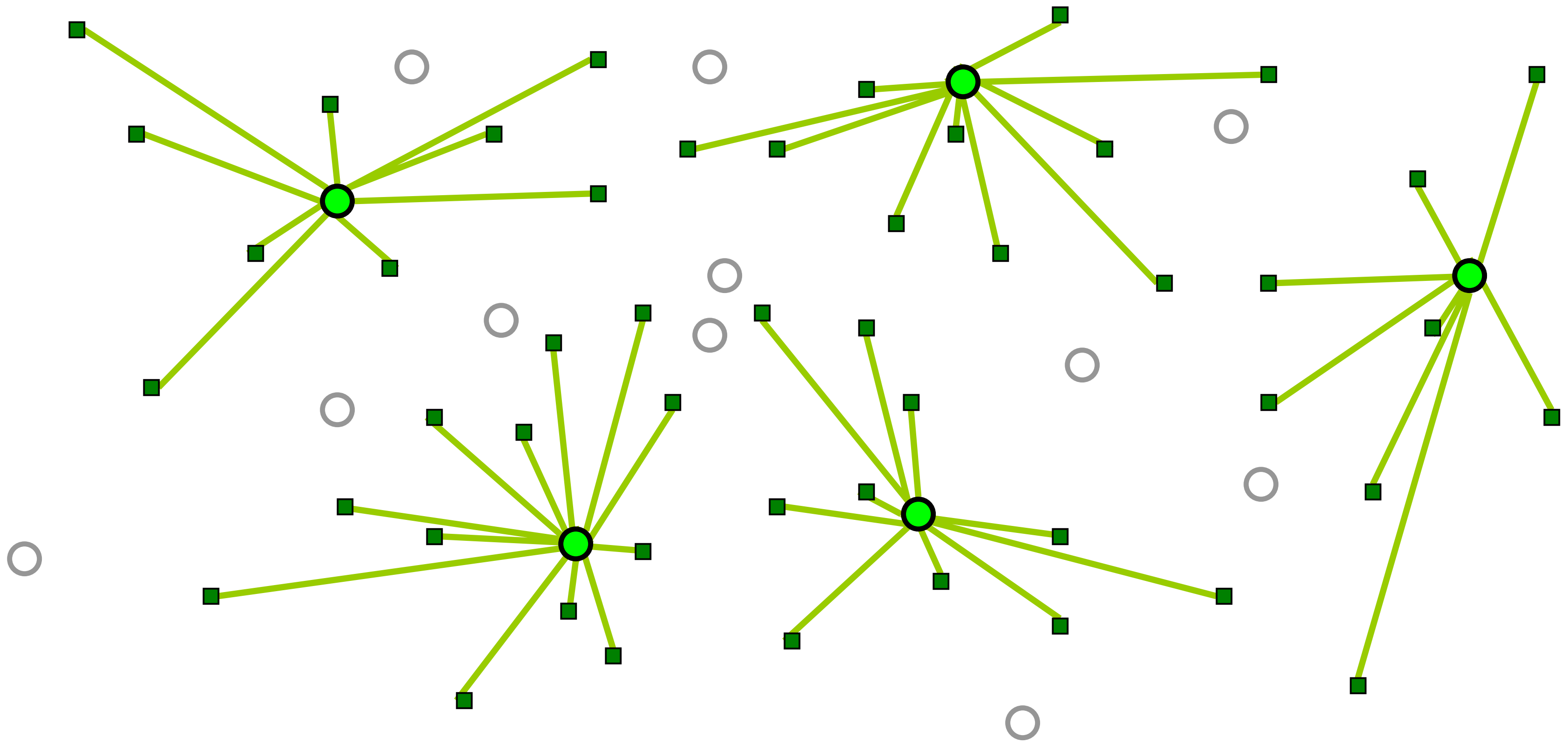
improved solution

Local Search - Example



improved solution
(with wrong assignments)

Local Search - Example



improved solution
(with proper assignments)

Local Search

- Introduced in [Teitz and Bart, 1968].
- Widely used in practice:
  - On its own:
    - [Whitaker, 1983];
    - [Rosing, 1997].
  - As a subroutine of metaheuristics:
    - [Rolland et al., 1996] - Tabu Search
    - [Voss, 1996] - “Reverse Elimination” (Tabu Search)
    - [Hansen and Mladenović, 1997] - VNS
    - [Rosing and ReVelle, 1997] - “Heuristic Concentration”
    - [Hansen et al., 2001] - VNDS

Previous Implementations

- Straightforward implementation:
  - For each candidate pair of facilities, compute profit:
    - $p(m-p) = O(pm)$ pairs;
    - $O(n)$ time to compute profit in each case;
    - $O(pmn)$ total time (cubic).
- In 1983, Whitaker proposed a much better implementation (named *Fast Interchange*).
- Key observation:
  - Given a candidate for insertion, the best removal can be computed in $O(n+m)$ time.
  - There are $O(m)$ candidates, so the overall running time is quadratic.

Our Implementation

- We propose another implementation:
  - same worst case complexity;
  - faster in practice, especially for large instances.
- Key idea: use information gathered in early iterations to speed up later ones.
  - Solution changes very little between iterations:
    - swap has a local effect.
  - Whitaker's implementation does not use this:
    - iterations are independent.
  - We use extra memory to avoid repeating previously executed calculations.

Deletion

- For each facility f_r in the solution, compute amount lost if it were deleted from the solution (and not replaced);
- That's the cost of transferring all facilities assigned to f_r to their second closest facilities:

$$loss(f_r) = \sum_{u:\phi_1(u)=f_r} [d(u, \phi_2(u)) - d(u, f_r)]$$

- Save the result: $loss$ is an array.

Notation:

– $\phi_1(u)$: facility in the solution that is closest to u;

– $\phi_2(u)$: second closest facility to u in the solution.

Insertion

- For each facility f_i not in the solution, compute amount gained if it were inserted (and no facility removed);
- That's the amount saved by transferring to f_i users that are closer to it than to their current facilities:

$$gain(f_i) = \sum_{u \in U} \max\{0, d(u, \phi_1(u)) - d(u, f_i)\}$$

- Save the result: $gain$ is also an array.

Swap

- We are interested in how profitable a *swap* is:

$$profit(f_i, f_r) = gain(f_i) - loss(f_r)$$

Swap

- We are interested in how profitable a *swap* is.
  - It would be nice if the profit were

$$profit(f_i, f_r) = gain(f_i) - loss(f_r)$$

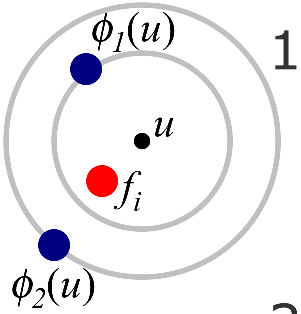
  - But it isn't: f_i and f_r "interact" with each other.
  - The correct expression is

$$profit(f_i, f_r) = gain(f_i) - loss(f_r) + extra(f_i, f_r)$$

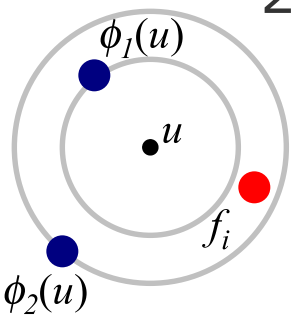
    (for a properly defined $extra$ function).
  - $extra$ can be thought of as a correction factor.

Correction Factor

- Things will "go wrong" for a user u iff:
  - f_r is the facility that is closest to u; **and**
  - One of two things happens:



1. The new facility is closer to u than $\phi_1(u)$ is.
   - When computing $loss$, we predicted that u would be reassigned to $\phi_2(u)$. This will not happen.
   - Loss overestimated by $[d(u, \phi_2(u)) - d(u, f_r)]$.

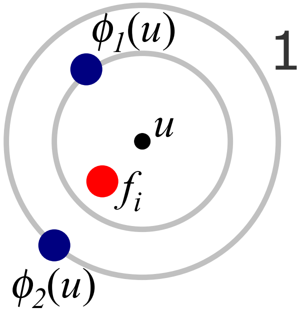


2. The new facility is farther to u than $\phi_1(u)$, but closer than $\phi_2(u)$.
   - When computing $loss$, we predicted that u would be reassigned to $\phi_2(u)$, but it should be reassigned to f_i.
   - Loss overestimated by $[d(u, \phi_2(u)) - d(u, f_i)]$.

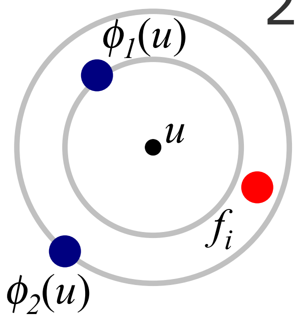
- Note that in both "wrong" cases we have overestimated the loss; $extra$ will be additive.

Correction Factor

- Things will "go wrong" for a user u iff:
  - f_r is the facility that is closest to u; **and**
  - One of two things happens:



1. The new facility is closer to u than $\phi_1(u)$ is.
   - Prediction: u will have to be reassigned to $\phi_2(u)$;
   - Fact: not necessary, $\phi_1(u)$ will take care of it.
   - Loss overestimated by $[d(u, \phi_2(u)) - d(u, f_r)]$.



2. The new facility is farther to u than $\phi_1(u)$, but closer than $\phi_2(u)$.
   - Prediction: u reassigned to $\phi_2(u)$;
   - Fact: u reassigned to f_i.
   - Loss overestimated by $[d(u, \phi_2(u)) - d(u, f_i)]$.

- Note that in both "wrong" cases we have overestimated the loss; *extra* will be additive.

Correction Factor

- From the conditions in the previous slide, we can determine what *extra* must be:

$$extra(f_i, f_r) = \sum_{\substack{u:\,[\phi_1(u)=f_r]\wedge \\ [d(u,\phi_1(u))\le d(u,f_i)<d(u,\phi_2(u))]}} [d(u,\phi_2(u)) - d(u,f_i)]$$

$$+ \sum_{\substack{u:\,[\phi_1(u)=f_r]\wedge \\ [d(u,f_i)<d(u,\phi_1(u))\le d(u,\phi_2(u))]}} [d(u,\phi_2(u)) - d(u,f_r)]$$

- Simplifying, we get

$$extra(f_i, f_r) = \sum_{\substack{u:\,[\phi_1(u)=f_r]\wedge \\ [d(u,f_i)<d(u,\phi_2(u))]}} [d(u,\phi_2(u)) - \max\{d(u,f_i), d(u,f_r)\}]$$

- This can be computed in $O(mn)$ time for all pairs.
- *extra* will be a matrix.

Our Implementation

- So we have to compute three structures:

$$loss(f_r) = \sum_{u:\phi_1(u)=f_r} [d(u,\phi_2(u)) - d(u,f_r)]$$

$$gain(f_i) = \sum_{u \in U} \max\{0, d(u,\phi_1(u)) - d(u,f_i)\}$$

$$extra(f_i, f_r) = \sum_{\substack{u:\ [\phi_1(u)=f_r] \wedge \\ [d(u,f_i)<d(u,\phi_2(u))]}} [d(u,\phi_2(u)) - \max\{d(u,f_i), d(u,f_r)\}]$$

- Each of them is a summation over the set of users:
  - We can compute the contribution of each user independently.

Our Implementation

```
function updateStructures (S,u,loss,gain,extra,φ1,φ2)
     f_r = φ1(u);
     loss[f_r] += d(u,φ2(u)) - d(u,φ1(u));
     forall (f_i∉S) do {
         if (d(u,f_i)<d(u,φ2(u))) then
                 gain[f_i] += max{0, d(u,φ1(u)) - d(u,f_i)};
                 extra[f_i,f_r] += d(u,φ2(u)) – max{d(u,f_i), d(u,f_r)};
         endif
     endforall
end updateStructures
```

- We can compute the contribution of each user independently.
- $O(m)$ time per user.

Our Implementation

- So each iteration of our method is as follows:
  1. Determine closeness information: $O(pm)$ time;
  2. Compute $gain$, $loss$, and $extra$: $O(mn)$ time;
  3. Use $gain$, $loss$, and $extra$ to find best swap: $O(pm)$ time.
- That's the same as Whitaker's implementation, but
  - much more complicated;
  - uses much more memory:
    - $extra$ is an $O(pm)$-sized matrix.
- Why would this be better?
  - Don't need to compute everything in every iteration;
  - we just need to update $gain$, $loss$, and $extra$;
  - only contributions of *affected users* are recomputed.

Our Implementation

```
function localSearch (S,φ1,φ2)
  A := U;
  resetStructures(gain,loss,extra);
  while (TRUE) do {
    forall (u∈A) do updateStructures (S,u,gain,loss,extra,φ1,φ2);
    (fr,fi,profit) := findBestNeighbor (gain,loss,extra);
    if (profit ≤ 0) then break;
    A := ∅;
    forall (u∈U) do
      if ((φ1(u)=fr) or (φ2(u)=fr) or (d(u,fi)<d(u,φ2(u)))) then
        A := A∪{u};
      endif;
    endforall
    forall (u∈A) do undoUpdateStructures(S,u,gain,loss,extra,φ1,φ2);
    insert(S,fi);
    remove(S,fr);
    updateClosest(S,fi,fr,φ1,φ2);
  endwhile
end localSearch
```

Our Implementation

```
function localSearch (S,φ1,φ2)
  A := U;
  resetStructures(gain,loss,extra);
  while (TRUE) do {
    forall (u∈A) do updateStructures (S,u,gain,loss,extra,φ1,φ2);
    (fr,fi,profit) := findBestNeighbor (gain,loss,extra);
    if (profit ≤ 0) then break;
    A := ∅;
    forall (u∈U) do
      if ((φ1(u)=fr) or (φ2(u)=fr) or (d(u,fi)<d(u,φ2(u)))) then
        A := A∪{u};
      endif;
    endforall
    forall (u∈A) do undoUpdateStructures(S,u,gain,loss,extra,φ1,φ2);
    insert(S,fi);
    remove(S,fr);
    updateClosest(S,fi,fr,φ1,φ2);
  endwhile
end localSearch
```

Input: solution to be changed and related closeness information.

Our Implementation

```
function localSearch (S,φ1,φ2)
  A := U;
  resetStructures(gain,loss,extra);
  while (TRUE) do {
    forall (u∈A) do updateStructures (S,u,gain,loss,extra,φ1,φ2);
    (fr,fi,profit) := findBestNeighbor (gain,loss,extra);
    if (profit ≤ 0) then break;
    A := ∅;
    forall (u∈U) do
      if ((φ1(u)=fr) or (φ2(u)=fr) or (d(u,fi)<d(u,φ2(u)))) then
        A := A∪{u};
      endif;
    endforall
    forall (u∈A) do undoUpdateStructures(S,u,gain,loss,extra,φ1,φ2);
    insert(S,fi);
    remove(S,fr);
    updateClosest(S,fi,fr,φ1,φ2);
  endwhile
end localSearch
```

All users affected in the beginning (*gain*, *loss*, and *extra* must be computed for all of them).

Our Implementation

```
function localSearch (S,φ1,φ2)
  A := U;
  resetStructures(gain,loss,extra);
  while (TRUE) do {
    forall (u∈A) do updateStructures (S,u,gain,loss,extra,φ1,φ2);
    (fr,fi,profit) := findBestNeighbor (gain,loss,extra);
    if (profit ≤ 0) then break;
    A := ∅;
    forall (u∈U) do
      if ((φ1(u)=fr) or (φ2(u)=fr) or (d(u,fi)<d(u,φ2(u)))) then
        A := A∪{u};
      endif;
    endforall
    forall (u∈A) do undoUpdateStructures(S,u,gain,loss,extra,φ1,φ2);
    insert(S,fi);
    remove(S,fr);
    updateClosest(S,fi,fr,φ1,φ2);
  endwhile
end localSearch
```

Initialize all positions of *gain*, *loss*, and *extra* to zero.

Our Implementation

Add contributions of all affected users to *gain*, *loss*, and *extra*.

```
function localSearch (S,φ1,φ2)
  A := U;
  resetStructures(gain,loss,extra);
  while (TRUE) do {
    forall (u∈A) do updateStructures (S,u,gain,loss,extra,φ1,φ2);
    (fr,fi,profit) := findBestNeighbor (gain,loss,extra);
    if (profit ≤ 0) then break;
    A := ∅;
    forall (u∈U) do
      if ((φ1(u)=fr) or (φ2(u)=fr) or (d(u,fi)<d(u,φ2(u)))) then
        A := A∪{u};
      endif;
    endforall
    forall (u∈A) do undoUpdateStructures(S,u,gain,loss,extra,φ1,φ2);
    insert(S,fi);
    remove(S,fr);
    updateClosest(S,fi,fr,φ1,φ2);
  endwhile
end localSearch
```

Our Implementation

```
function localSearch (S,φ1,φ2)
  A := U;
  resetStructures(gain,loss,extra);
  while (TRUE) do {
    forall (u∈A) do updateStructures (S,u,gain,loss,extra,φ1,φ2);
    (fr,fi,profit) := findBestNeighbor (gain,loss,extra);
    if (profit ≤ 0) then break;
    A := ∅;
    forall (u∈U) do
      if ((φ1(u)=fr) or (φ2(u)=fr) or (d(u,fi)<d(u,φ2(u)))) then
        A := A∪{u};
      endif;
    endforall
    forall (u∈A) do undoUpdateStructures(S,u,gain,loss,extra,φ1,φ2);
    insert(S,fi);
    remove(S,fr);
    updateClosest(S,fi,fr,φ1,φ2);
  endwhile
end localSearch
```

Determine the best swap to make.

Our Implementation

```
function localSearch (S,φ1,φ2)
  A := U;
  resetStructures(gain,loss,extra);
  while (TRUE) do {
    forall (u∈A) do updateStructures (S,u,gain,loss,extra,φ1,φ2);
    (fr,fi,profit) := findBestNeighbor (gain,loss,extra);
    if (profit ≤ 0) then break;
    A := ∅;
    forall (u∈U) do
      if ((φ1(u)=fr) or (φ2(u)=fr) or (d(u,fi)<d(u,φ2(u)))) then
        A := A∪{u};
      endif;
    endforall
    forall (u∈A) do undoUpdateStructures(S,u,gain,loss,extra,φ1,φ2);
    insert(S,fi);
    remove(S,fr);
    updateClosest(S,fi,fr,φ1,φ2);
  endwhile
end localSearch
```

Swap will be performed only if profitable.

Our Implementation

```
function localSearch (S,φ1,φ2)
  A := U;
  resetStructures(gain,loss,extra);
  while (TRUE) do {
    forall (u∈A) do updateStructures (S,u,gain,loss,extra,φ1,φ2);
    (fr,fi,profit) := findBestNeighbor (gain,loss,extra);
    if (profit ≤ 0) then break;
    A := ∅;
    forall (u∈U) do
      if ((φ1(u)=fr) or (φ2(u)=fr) or (d(u,fi)<d(u,φ2(u)))) then
        A := A∪{u};
      endif;
    endforall
    forall (u∈A) do undoUpdateStructures(S,u,gain,loss,extra,φ1,φ2);
    insert(S,fi);
    remove(S,fr);
    updateClosest(S,fi,fr,φ1,φ2);
  endwhile
end localSearch
```

Determine which users will be affected (those who are close to at least one of the facilities involved in the swap).

Our Implementation

```
function localSearch (S,φ1,φ2)
  A := U;
  resetStructures(gain,loss,extra);
  while (TRUE) do {
    forall (u∈A) do updateStructures (S,u,gain,loss,extra,φ1,φ2);
    (fr,fi,profit) := findBestNeighbor (gain,loss,extra);
    if (profit ≤ 0) then break;
    A := ∅;
    forall (u∈U) do
      if ((φ1(u)=fr) or (φ2(u)=fr) or (d(u,fi)<d(u,φ2(u)))) then
        A := A∪{u};
      endif;
    endforall
    forall (u∈A) do undoUpdateStructures(S,u,gain,loss,extra,φ1,φ2);
    insert(S,fi);
    remove(S,fr);
    updateClosest(S,fi,fr,φ1,φ2);
  endwhile
end localSearch
```

Disregard previous contributions from affected users to *gain*, *loss*, and *extra*.

Our Implementation

```
function localSearch (S,φ1,φ2)
  A := U;
  resetStructures(gain,loss,extra);
  while (TRUE) do {
    forall (u∈A) do updateStructures (S,u,gain,loss,extra,φ1,φ2);
    (fr,fi,profit) := findBestNeighbor (gain,loss,extra);
    if (profit ≤ 0) then break;
    A := ∅;
    forall (u∈U) do
      if ((φ1(u)=fr) or (φ2(u)=fr) or (d(u,fi)<d(u,φ2(u)))) then
        A := A∪{u};
      endif;
    endforall
    forall (u∈A) do undoUpdateStructures(S,u,gain,loss,extra,φ1,φ2);
    insert(S,fi);
    remove(S,fr);
    updateClosest(S,fi,fr,φ1,φ2);
  endwhile
end localSearch
```

Finally, perform the swap.

Our Implementation

```
function localSearch (S,φ1,φ2)
  A := U;
  resetStructures(gain,loss,extra);
  while (TRUE) do {
    forall (u∈A) do updateStructures (S,u,gain,loss,extra,φ1,φ2);
    (fr,fi,profit) := findBestNeighbor (gain,loss,extra);
    if (profit ≤ 0) then break;
    A := ∅;
    forall (u∈U) do
      if ((φ1(u)=fr) or (φ2(u)=fr) or (d(u,fi)<d(u,φ2(u)))) then
        A := A∪{u};
      endif;
    endforall
    forall (u∈A) do undoUpdateStructures(S,u,gain,loss,extra,φ1,φ2);
    insert(S,fi);
    remove(S,fr);
    updateClosest(S,fi,fr,φ1,φ2);
  endwhile
end localSearch
```

Update closeness information for next iteration.

Bottlenecks

```
function localSearch (S,φ1,φ2)
  A := U;
  resetStructures(gain,loss,extra);
  while (TRUE) do {
3   forall (u∈A) do updateStructures (S,u,gain,loss,extra,φ1,φ2);
2   (fr,fi,profit) := findBestNeighbor (gain,loss,extra);
    if (profit ≤ 0) then break;
    A := ∅;
    forall (u∈U) do
      if ((φ1(u)=fr) or (φ2(u)=fr) or (d(u,fi)<d(u,φ2(u)))) then
        A := A∪{u};
      endif;
    endforall
3   forall (u∈A) do undoUpdateStructures(S,u,gain,loss,extra,φ1,φ2);
    insert(S,fi);
    remove(S,fr);
1   updateClosest(S,fi,fr,φ1,φ2);
  endwhile
end localSearch
```

1. Updating closeness information;
2. finding the best swap to make;
3. updating auxiliary structures.

Bottleneck 1 – Closeness

```
function localSearch (S,φ1,φ2)
  A := U;
  resetStructures(gain,loss,extra);
  while (TRUE) do {
     forall (u∈A) do updateStructures (S,u,gain,loss,extra,φ1,φ2);
     (fr,fi,profit) := findBestNeighbor (gain,loss,extra);
     if (profit ≤ 0) then break;
     A := ∅;
     forall (u∈U) do
        if ((φ1(u)=fr) or (φ2(u)=fr) or (d(u,fi)<d(u,φ2(u)))) then
           A := A∪{u};
        endif;
     endforall
     forall (u∈A) do undoUpdateStructures(S,u,gain,loss,extra,φ1,φ2);
     insert(S,fi);
     remove(S,fr);
     updateClosest(S,fi,fr,φ1,φ2);
  endwhile
end localSearch
```

Bottleneck 1 – Closeness

- Two kinds of change may occur with a user:
  1. The new facility (f_i) becomes its closest or second closest facility:
     - Update takes constant time.
  2. The facility removed (f_r) was the user's closest or second closest:
     - Need to look for a new second closest;
     - Takes O(p) time.
- The second case could be a bottleneck, but in practice only a few users fall into this case.
  - Only these need to be tested.
  - [Hansen and Mladenović, 1997].

Bottleneck 2 – Best Neighbor

```
function localSearch (S,φ1,φ2)
  A := U;
  resetStructures(gain,loss,extra);
  while (TRUE) do {
    forall (u∈A) do updateStructures (S,u,gain,loss,extra,φ1,φ2);
    (fr,fi,profit) := findBestNeighbor (gain,loss,extra);
    if (profit ≤ 0) then break;
    A := ∅;
    forall (u∈U) do
      if ((φ1(u)=fr) or (φ2(u)=fr) or (d(u,fi)<d(u,φ2(u)))) then
        A := A∪{u};
      endif;
    endforall
    forall (u∈A) do undoUpdateStructures(S,u,gain,loss,extra,φ1,φ2);
    insert(S,fi);
    remove(S,fr);
    updateClosest(S,fi,fr,φ1,φ2);
  endwhile
end localSearch
```

Bottleneck 2 – Best Neighbor

- Number of potential swaps: *p(m-p).*
- Straightforward way to compute the best one:
  - Compute $profit(f_i, f_r)$ for all pairs and pick minimum:

$$profit(f_i, f_r) = gain(f_i) - loss(f_r) + extra(f_i, f_r)$$

  - This requires *O(mp)* time.
- Alternative:
  - As the initial candidate, pick the f_i with the largest $gain$ and the f_r with the smallest $loss$.
    - The best swap is at least as good as this.
      - Reason: $extra$ is always nonnegative.
  - Compute the exact $profit$ only for pairs that have $extra$ greater than zero.

Bottleneck 2 – Best Neighbor

- Worst case:
  - $O(pm)$ (exactly the same)
- In practice:
  - $extra(f_i, f_r)$ represents the "interference" between these two facilities.
  - Local phenomenon: each facility interacts with some facilities nearby.
  - $extra$ is likely to have very few nonzero elements, especially when p is large.
- Use sparse matrix representation for $extra$:
  - each row represented as a linked list of nonzero elements.
  - "side effect": less memory (usually).

Bottleneck 3 – Updating Structures

```
function localSearch (S,φ1,φ2)
  A := U;
  resetStructures(gain,loss,extra);
  while (TRUE) do {
    forall (u∈A) do updateStructures (S,u,gain,loss,extra,φ1,φ2);
     (fr,fi,profit) := findBestNeighbor (gain,loss,extra);
     if (profit ≤ 0) then break;
     A := ∅;
     forall (u∈U) do
       if ((φ1(u)=fr) or (φ2(u)=fr) or (d(u,fi)<d(u,φ2(u)))) then
         A := A∪{u};
       endif;
    endforall
    forall (u∈A) do undoUpdateStructures(S,u,gain,loss,extra,φ1,φ2);
    insert(S,fi);
    remove(S,fr);
    updateClosest(S,fi,fr,φ1,φ2);
  endwhile
end localSearch
```

Bottleneck 3 – Updating Structures

```
function updateStructures (S,u,loss,gain,extra,φ1,φ2)
    fr = φ1(u);
    loss[fr] += d(u,φ2(u)) - d(u,φ1(u));
    forall (fi∉S) do
        if (d(u,fi)<d(u,φ2(u))) then
            gain[fi] += max{0, d(u,φ1(u)) - d(u,fi)};
            extra[fi,fr] += d(u,φ2(u)) - max{d(u,fi), d(u,fr)};
        endif
    endforall
end updateStructures
```

This loop always takes *m-p* iterations.

Bottleneck 3 – Updating Structures

```
function updateStructures (S,u,loss,gain,extra,φ1,φ2)
    fr = φ1(u);
    loss[fr] += d(u,φ2(u)) - d(u,φ1(u));
    forall (fi∉S such that d(u,fi)<d(u,φ2(u))) do
            gain[fi] += max{0, d(u,φ1(u)) - d(u,fi)};
            extra[fi,fr] += d(u,φ2(u)) – max{d(u,fi), d(u,fr)};
    endforall
end updateStructures
```

We actually need only facilities that are very close to u.

– Preprocessing step:
  - for each user, sort all facilities in increasing order by distance (and keep the resulting list);
  - in the function above, we just need to check the appropriate prefix of the list.

Bottleneck 3: Updating Structures

- Preprocessing step:
  - Time:
    - $O(nm \log m)$;
    - preprocessing step executed only once, even if local search is run several times.
  - Space:
    - $O(mn)$ memory positions, which can be too much.
    - Alternative:
      - Keep only a prefix of the list (the closest facilities).
      - Use list as a cache:
        - » If enough elements present, use it;
        - » Otherwise, do as before: check all facilities.
      - Same worst case.

Results

- Three classes of instances:
  - ORLIB (sparse graphs):
    - 100 to 900 users, p between 5 and 200;
    - Distances given by shortest paths in the graph.
  - RW (random instances):
    - 100 to 1000 users, p between 10 and $n/2$;
    - Distances picked at random from $[1,n]$.
  - TSP (points on the plane):
    - 1400, 3038, or 5934 users, p between 10 and $n/3$;
    - Distances are Euclidean.

- In all cases, number of users is equal to the number of potential facilities.

Results

- Three variations analyzed:
  - **FM**: **F**ull **M**atrix, no preprocessing;
  - **SM**: **S**parse **M**atrix, no preprocessing;
  - **SMP**: **S**parse **M**atrix, with **P**reprocessing.
- These were run on all instances and compared to Whitaker's *fast interchange* method (**FI**).
  - As implemented in [Hansen and Mladenović, 1997].
- All methods (including **FI**) use the "smart" update of closeness information.
- Measure of relative performance: *speedup.*
  - Ratio between the running time of **FI** and the running time of our method.
  - All methods start from the same (greedy) solution.

Results

- Mean speedups when compared to Whitaker's **FI**:

Method	Description	ORLIB	RW	TSP
FM	full matrix, no preprocessing	3.0	4.1	11.7

  - Even our simplest variation is faster in practice;
  - Updating only *affected users* does pay off;
  - Speedups greater for larger instances.

Results

- Mean speedups when compared to Whitaker's **FI**:

Method	Description	ORLIB	RW	TSP
FM	full matrix, no preprocessing	3.0	4.1	11.7
SM	sparse matrix, no preprocessing	3.1	5.3	26.2

  - Checking only the nonzero elements of the *extra* matrix gives an additional speedup.
  - Again, better for larger instances.

Results

- Mean speedups when compared to Whitaker's **FI**:

Method	Description	ORLIB	RW	TSP
FM	full matrix, no preprocessing	3.0	4.1	11.7
SM	sparse matrix, no preprocessing	3.1	5.3	26.2
SMP	sparse matrix, full preprocessing	1.2	2.1	20.3

- – Preprocessing appears to be a little too expensive.
  - Still much faster than the original implementation.
- – But remember that preprocessing must be run just once, even if the local search is run more than once.

Results

- Mean speedups when compared to Whitaker's **FI**:

Method	Description	ORLIB	RW	TSP
FM	full matrix, no preprocessing	3.0	4.1	11.7
SM	sparse matrix, no preprocessing	3.1	5.3	26.2
SMP	sparse matrix, full preprocessing	1.2	2.1	20.3
SMP*	sparse matrix, full preprocessing	8.7	15.1	177.6

(in **SMP***, preprocessing times are not included)

  - If we are able to amortize away the preprocessing time, significantly greater speedups are observed on average.
  - Typical case in metaheuristics.

Results

- Speedups w.r.t. Whitaker's **FI** (best cases):

Method	Description	ORLIB	RW	TSP
FM	full matrix, no preprocessing	12.7	12.4	31.1
SM	sparse matrix, no preprocessing	17.2	32.4	147.7
SMP	sparse matrix, full preprocessing	7.5	9.6	79.2
SMP*	sparse matrix, full preprocessing	67.0	113.9	862.1

(in **SMP***, preprocessing times are not included)

  - Speedups of up to three orders of magnitude were observed.
  - Greater for large instances with large values of p.

Results

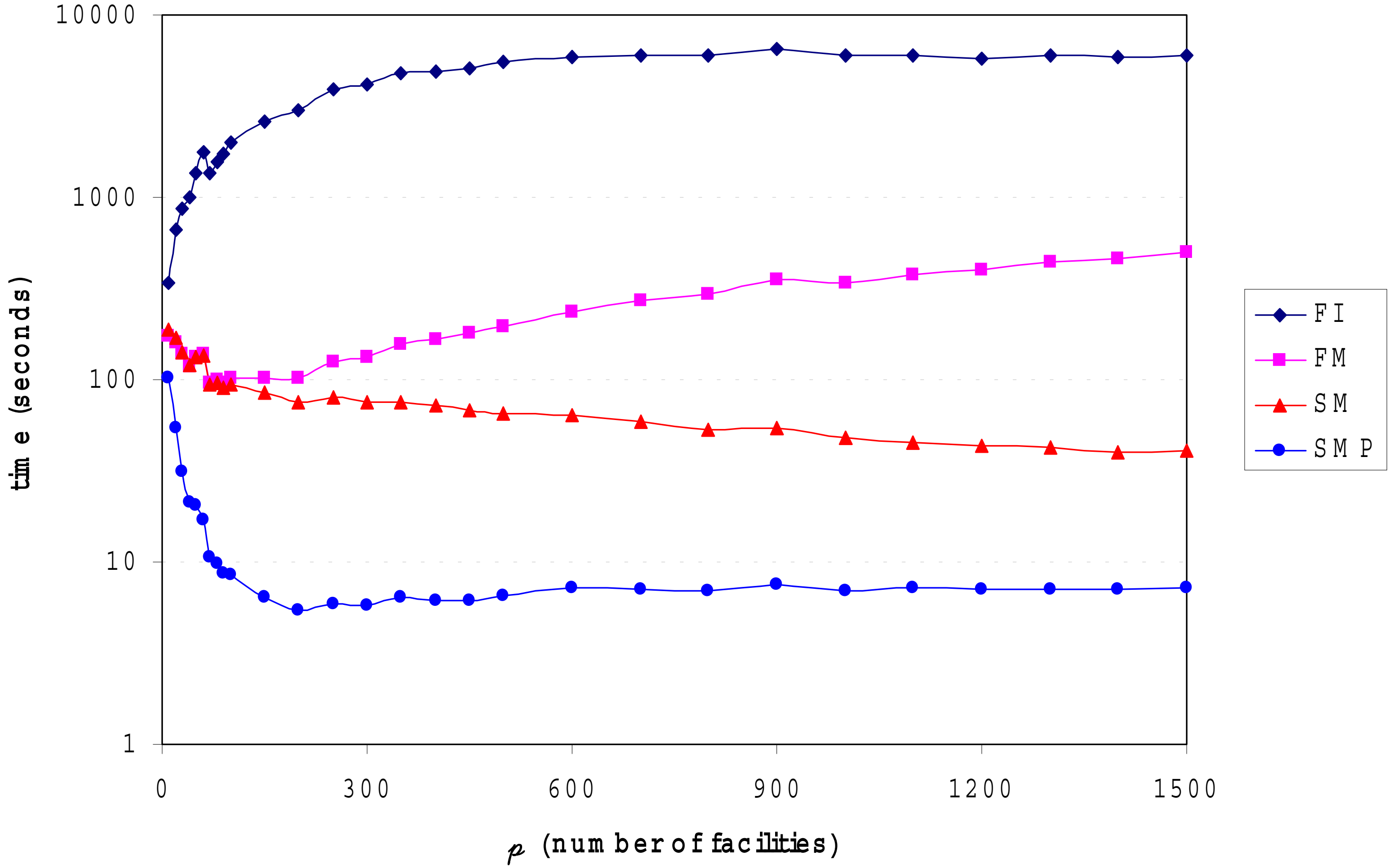
- Speedups w.r.t. Whitaker's **FI** (worst cases):

Method	Description	ORLIB	RW	TSP
FM	full matrix, no preprocessing	0.84	0.88	1.85
SM	sparse matrix, no preprocessing	0.74	0.75	1.72
SMP	sparse matrix, full preprocessing	0.22	0.18	1.33
SMP*	sparse matrix, full preprocessing	1.30	1.40	3.27

(in **SMP***, preprocessing times are not included)

  – For small instances, our method can be slower than Whitaker's; our constants are higher.
  – Once preprocessing times are amortized, even that does not happen.

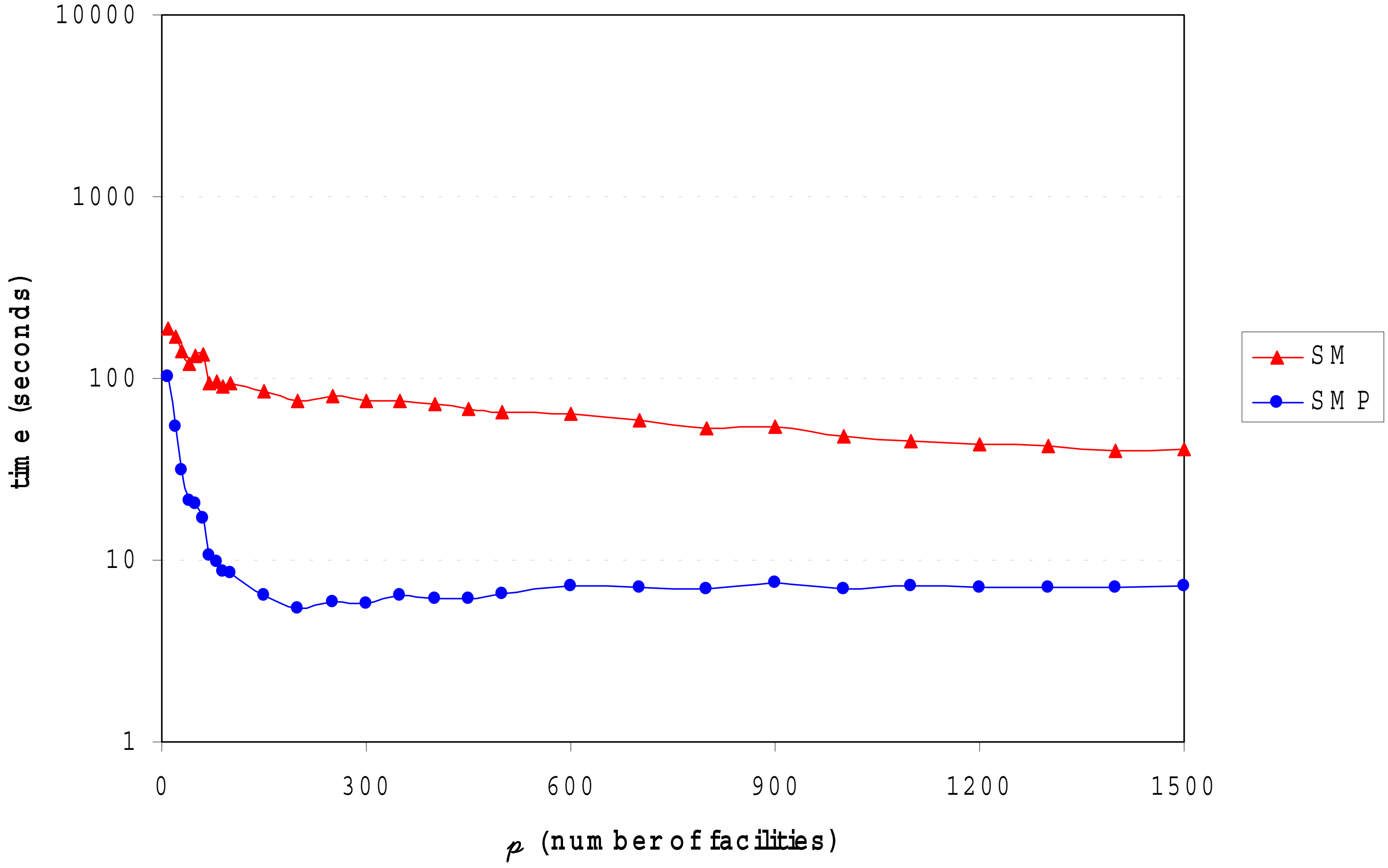
Results



Largest instance tested: 5934 users, Euclidean.

(preprocessing times not considered)

Results

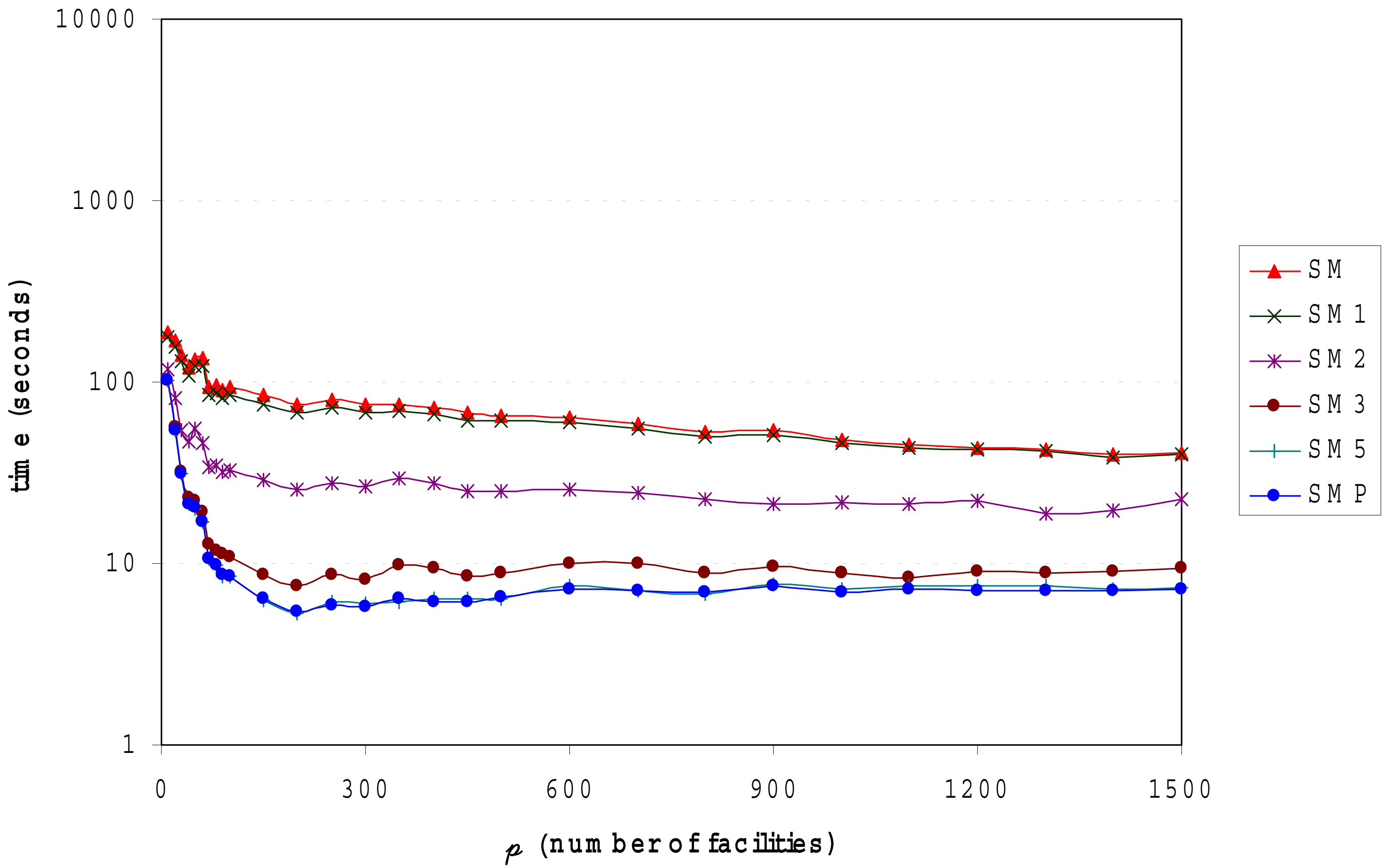


Note that preprocessing significantly accelerates the algorithm.

Results

- Preprocessing greatly accelerates the algorithm.
- However, it requires a great amount of memory:
  - *n* lists of size *m*.
- We can make only partial lists.
  - We would like each list to the second closest open facility as often as possible:
  - the larger *m* is, the larger the list needs to be;
  - the larger *p* is, the smallest the list needs to be.
- Method **SM*q***:
  - Each user has a list of size $q\ m/p$.
  - Example: $m = 6000$, $p = 300$, $q = 5$.
    - Each user keeps a list of size 100;
    - in the "full" version, the list would have size 6000.

Results



For this instance, q=5 is already as fast as the full version.

Final Remarks

- New implementation of well-known local search.
- Uses extra memory, but much faster in practice.
- Accelerations are metric-independent.
- Especially useful for metaheuristics:
  - We have implemented a GRASP based on this local search with very promising results.
  - Other existing methods may benefit from it.
- There is still room for improvement:
  - metric-specific techniques (graphs, Euclidean);
  - perform preprocessing “on demand”.

The End

The End

Straightforward Implementation

- Straightforward Implementation:
  - For each candidate pair (f_i, f_r) of facilities, compute the profit that would be obtained:

$$profit(f_i, f_r) = \sum_{u:\phi_1(u) \neq f_r} \max\{0, [d(u, \phi_1(u)) - d(u, f_i)]\}$$
$$- \sum_{u:\phi_1(u) = f_r} [\min\{d(u, \phi_2(u)), d(u, f_i)\} - d(u, \phi_1(u))]$$

  - Notation:
    - $\phi_1(u)$: facility in the solution that is closest to u;
    - $\phi_2(u)$: second closest facility to u in the solution.

Straightforward Implementation

- Straightforward Implementation:
  - For each candidate pair (f_i, f_r) of facilities, compute the profit that would be obtained:

$$profit(f_i, f_r) = \boxed{\sum_{u:\phi_1(u) \neq f_r} \max\{0, [d(u,\phi_1(u)) - d(u, f_i)]\}} - \sum_{u:\phi_1(u) = f_r} [\min\{d(u,\phi_2(u)), d(u, f_i)\} - d(u,\phi_1(u))]$$

Gain from reassigning users to f_i, the new facility

Straightforward Implementation

- Straightforward Implementation:
  - For each candidate pair (f_i, f_r) of facilities, compute the profit that would be obtained:

$$profit(f_i, f_r) = \sum_{u:\phi_1(u)\neq f_r} \max\{0, [d(u,\phi_1(u)) - d(u, f_i)]\}$$
$$- \boxed{\sum_{u:\phi_1(u)=f_r} [\min\{d(u,\phi_2(u)), d(u, f_i)\} - d(u,\phi_1(u))]}$$

Loss from reassigning users previously assigned to f_r

Straightforward Implementation

- Straightforward Implementation:
  - For each candidate pair of facilities, compute the corresponding profit

$$profit(f_i, f_r) = \sum_{u:\phi_1(u)\neq f_r} \max\{0, [d(u,\phi_1(u)) - d(u,f_i)]\} - \sum_{u:\phi_1(u)=f_r} [\min\{d(u,\phi_2(u)), d(u,f_i)\} - d(u,\phi_1(u))]$$

  - Running time:
    - $O(pn)$ time to compute $\phi_1(u)$ and $\phi_2(u)$ for all u;
    - $p(m\text{-}p) = O(pm)$ candidate pairs;
    - $O(n)$ time to process each of them;
    - $O(pmn)$ total time.

Whitaker's Implementation

```
function findOut (S, f_i, φ_1, φ_2)
    w := 0;
    forall (f_r ∈ S) do v(f_r) := 0;
    forall (u ∈ U) do {
        if (d(u, f_i) < d(u, φ_1(u))) then
                w += d(u, φ_1(u)) - d(u, f_i);
        else
                v(φ_1(u)) += min{d(u, f_i), d(u, φ_2(u))} - d(u, φ_1(u));
        endif
    endforall
    f_r := argmin_{f∈S}{v(f)};
    profit := w - v(f_r);
    return (f_r, profit);
end findOut;
```

Whitaker's Implementation

```
function findOut (S, f_i, φ_1, φ_2)
    w := 0;
    forall (f_r ∈ S) do v(f_r) := 0;
    forall (u ∈ U) do {
        if (d(u,f_i) < d(u,φ_1(u))) then
                w += d(u,φ_1(u)) - d(u,f_i);
        else
                v(φ_1(u)) += min{d(u,f_i),d(u,φ_2(u))} - d(u,φ_1(u));
        endif
    endforall
    f_r := argmin_{f∈S}{v(f)};
    profit := w - v(f_r);
    return (f_r, profit);
end findOut;
```

Input: current solution, facility to insert, closeness information

- Notation:
  - $\phi_1(u)$: facility in the solution that is closest to u;
  - $\phi_2(u)$: second closest facility to u in the solution.

Whitaker's Implementation

```
function findOut (S, f_i, φ_1, φ_2)
     w := 0;
     forall (f_r ∈ S) do v(f_r) := 0;
     forall (u ∈ U) do {
         if (d(u, f_i) < d(u, φ_1(u))) then
                 w += d(u, φ_1(u)) - d(u, f_i);
         else
                 v(φ_1(u)) += min{d(u, f_i), d(u, φ_2(u))} - d(u, φ_1(u));
         endif
     endforall
     f_r := argmin_{f ∈ S}{v(f)};
     profit := w - v(f_r);
     return (f_r, profit);
end findOut;
```

Output: facility to remove and associated profit (may be negative)

Whitaker's Implementation

```
function findOut (S, f_i, φ_1, φ_2)
    w := 0;
    forall (f_r ∈ S) do v(f_r) := 0;
    forall (u ∈ U) do {
        if (d(u,f_i) < d(u,φ_1(u))) then
                w += d(u,φ_1(u)) - d(u,f_i);
        else
                v(φ_1(u)) += min{d(u,f_i), d(u,φ_2(u))} - d(u,φ_1(u));
        endif
    endforall
    f_r := argmin_{f∈S}{v(f)};
    profit := w - v(f_r);
    return (f_r, profit);
end findOut;
```

This variable will account for the total gain due to reassigning users to the new facility.

Whitaker's Implementation

```
function findOut (S, f_i, φ_1, φ_2)
    w := 0;
    forall (f_r ∈ S) do v(f_r) := 0;
    forall (u ∈ U) do {
        if (d(u,f_i) < d(u,φ_1(u))) then
                w += d(u,φ_1(u)) - d(u,f_i);
        else
                v(φ_1(u)) += min{d(u,f_i),d(u,φ_2(u))} - d(u,φ_1(u));
        endif
    endforall
    f_r := argmin_{f∈S}{v(f)};
    profit := w - v(f_r);
    return (f_r, profit);
end findOut;
```

Variable $v(f_r)$ represents how much would be lost if f_r were removed from the solution.

Whitaker's Implementation

```
function findOut (S, f_i, φ_1, φ_2)
    w := 0;
    forall (f_r ∈ S) do v(f_r) := 0;
    forall (u ∈ U) do {
        if (d(u,f_i) < d(u,φ_1(u))) then
                w += d(u,φ_1(u)) - d(u,f_i);
        else
                v(φ_1(u)) += min{d(u,f_i),d(u,φ_2(u))} - d(u,φ_1(u));
        endif
    endforall
    f_r := argmin_{f∈S}{v(f)};
    profit := w - v(f_r);
    return (f_r, profit);
end findOut;
```

Case 1: User wants to be reassigned to the new facility. Compute the profit.

Whitaker's Implementation

```
function findOut (S,f_i,φ_1,φ_2)
    w := 0;
    forall (f_r∈S) do v(f_r) := 0;
    forall (u∈U) do {
        if (d(u,f_i)<d(u,φ_1(u))) then
                w += d(u,φ_1(u)) - d(u,f_i);
        else
                v(φ_1(u)) += min{d(u,f_i),d(u,φ_2(u))} - d(u,φ_1(u));
        endif
    endforall
    f_r := argmin_{f∈S}{v(f)};
    profit := w - v(f_r);
    return (f_r,profit);
end findOut;
```

Case 2: User does not want to be reassigned. We compute the cost of reassigning if we have to remove its closest facility.

Whitaker's Implementation

```
function findOut (S,f_i,φ_1,φ_2)
    w := 0;
    forall (f_r∈S) do v(f_r) := 0;
    forall (u∈U) do {
        if (d(u,f_i)<d(u,φ_1(u))) then
                w += d(u,φ_1(u)) - d(u,f_i);
        else
                v(φ_1(u)) += min{d(u,f_i),d(u,φ_2(u))} - d(u,φ_1(u));
        endif
    endforall
    f_r := argmin_{f∈S}{v(f)};
    profit := w - v(f_r);
    return (f_r,profit);
end findOut;
```

Pick the facility with the smallest reassignment cost and compute the "real" profit associated with it.

Whitaker's Implementation

```
function findOut (S, f_i, φ_1, φ_2)
     w := 0;
     forall (f_r ∈ S) do v(f_r) := 0;
     forall (u ∈ U) do {
        if (d(u,f_i) < d(u,φ_1(u))) then
                w += d(u,φ_1(u)) - d(u,f_i);
        else
                v(φ_1(u)) += min{d(u,f_i),d(u,φ_2(u))} - d(u,φ_1(u));
        endif
     endforall
     f_r := argmin_{f∈S}{v(f)};
     profit := w - v(f_r);
     return (f_r,profit);
end findOut;
```

This procedure takes $O(n+m)$ time.

Our Implementation

- For each facility, compute the following values:
  - $loss(f_r)$: amount lost if f_r were removed from the solution (no facility inserted):

$$loss(f_r) = \sum_{u:\phi_1(u)=f_r} [d(u,\phi_2(u)) - d(u,f_r)]$$

    (users reassigned to second closest facilities)

  - $gain(f_i)$: how much is gained if f_i were inserted into the solution (no facility removed):

$$gain(f_i) = \sum_{u \in U} \max\{0, d(u,\phi_1(u)) - d(u,f_i)\}$$

    (close enough users would reassigned to f_i)

Results

- Variant *FM*:
  - full matrix;
  - no preprocessing.

- Speedups when compared to Whitaker's *FI*:

Class	*Best*	*Mean*	*Worst*
ORLIB	12.72	3.01	0.84
RW	12.42	4.14	0.88
TSP	31.14	11.68	1.85

Results

- Variant *SM*:
  - sparse matrix;
  - no preprocessing.

- Speedups when compared to Whitaker's *FI*:

Class	*Best*	*Mean*	*Worst*
ORLIB	17.21	3.10	0.74
RW	32.39	5.26	0.75
TSP	147.71	26.18	1.72

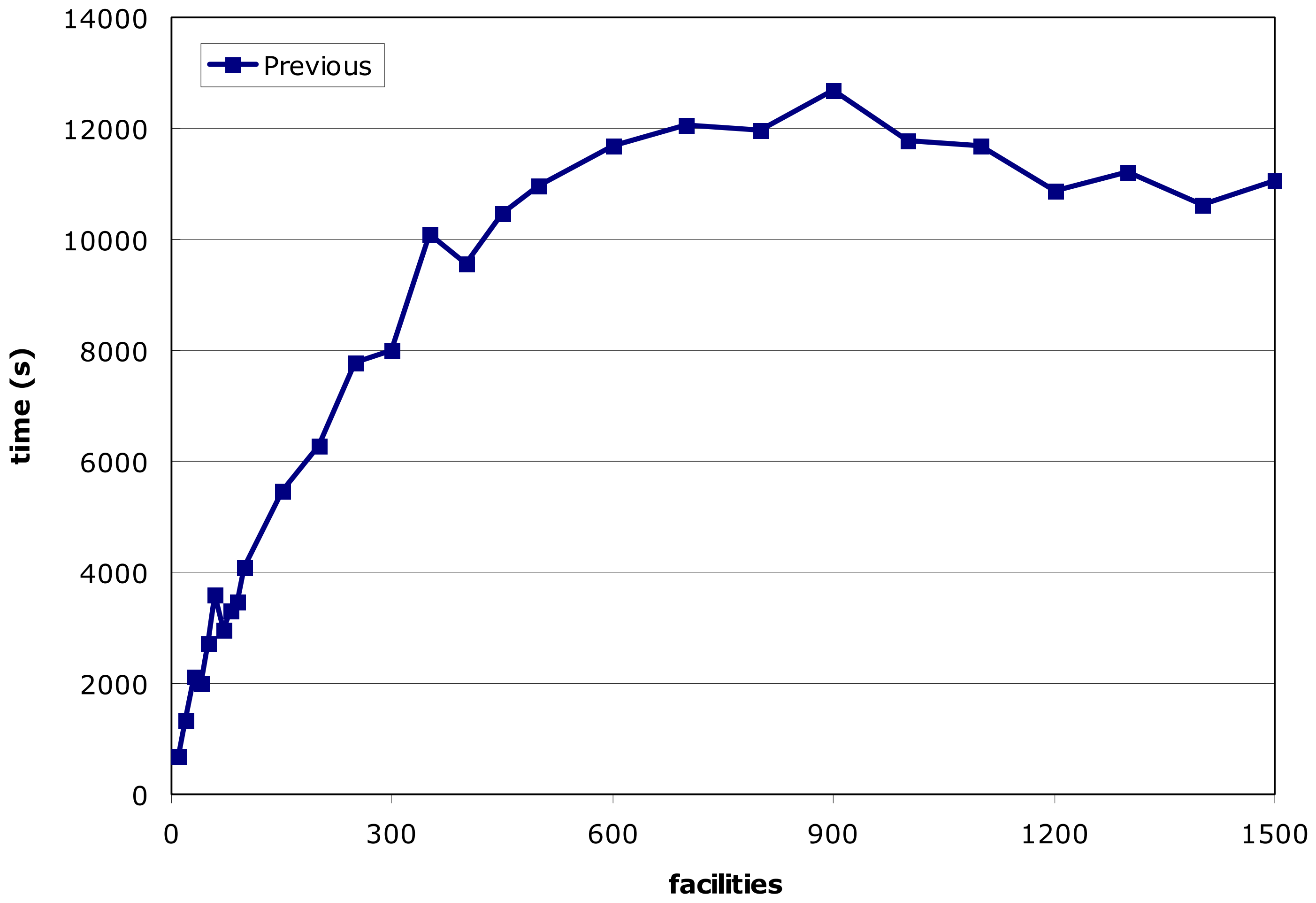
Results

- Variant *SMP*:
  - sparse matrix;
  - full preprocessing (complete list for each user)

- Speedups when compared to Whitaker's *FI*:

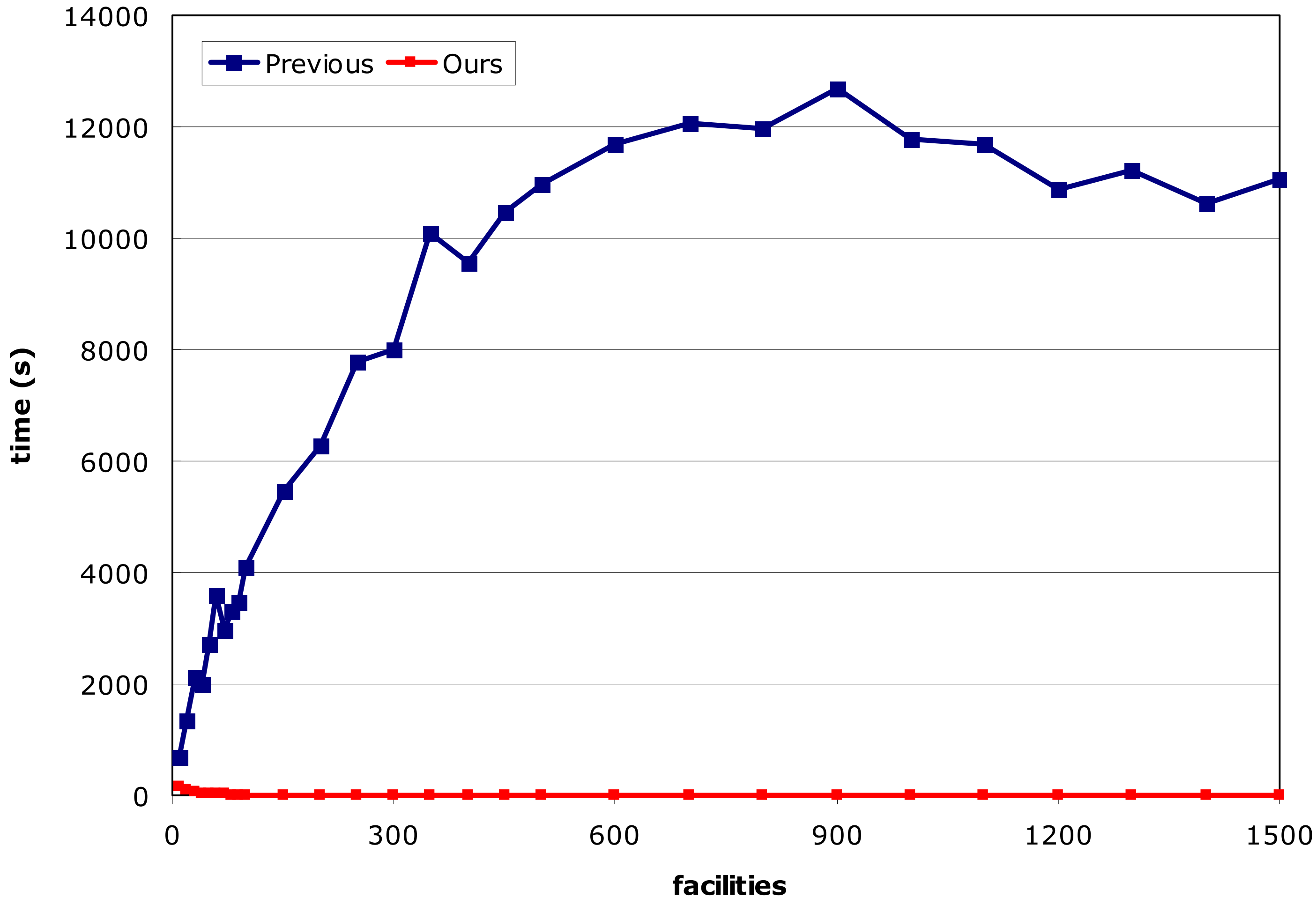
Class	*Local Search Only*			*Incl. Preprocessing*		
	Best	*Mean*	*Worst*	*Best*	*Mean*	*Worst*
ORLIB	67.0	8.7	1.30	7.5	1.2	0.22
RW	113.9	15.1	1.40	9.6	2.1	0.18
TSP	862.1	177.6	3.27	79.2	20.3	1.33

Local Search



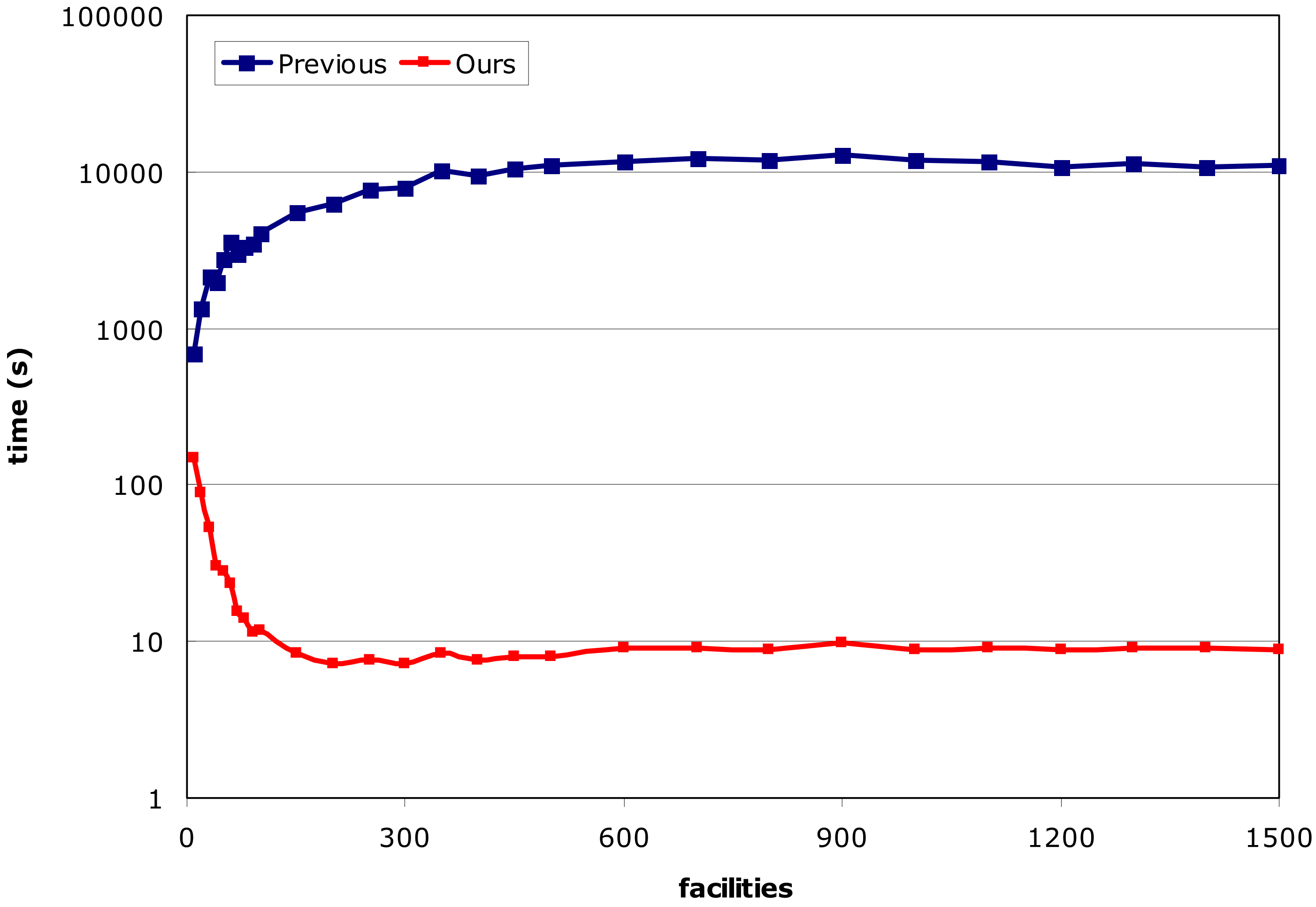
Euclidean instance, 5934 users/facilities

Local Search



Euclidean instance, 5934 users/facilities

Local Search



(replay in log scale...)

Results

- Tested on Euclidean and graph instances.
- Compares favorably with the 3 best heuristics available (within similar running times).
- Solution quality:
  - Worst: 0.12% above best solution known.
  - Best: improved best known by 1.397%.
- We are still working on improvements.