Finding approximate solutions for the p-median problem

Mauricio G. C. Resende

Algorithms & Optimization Research

AT&T Labs Research

Florham Park, New Jersey

mgcr@att.com

http://www.research.att.com/~mgcr

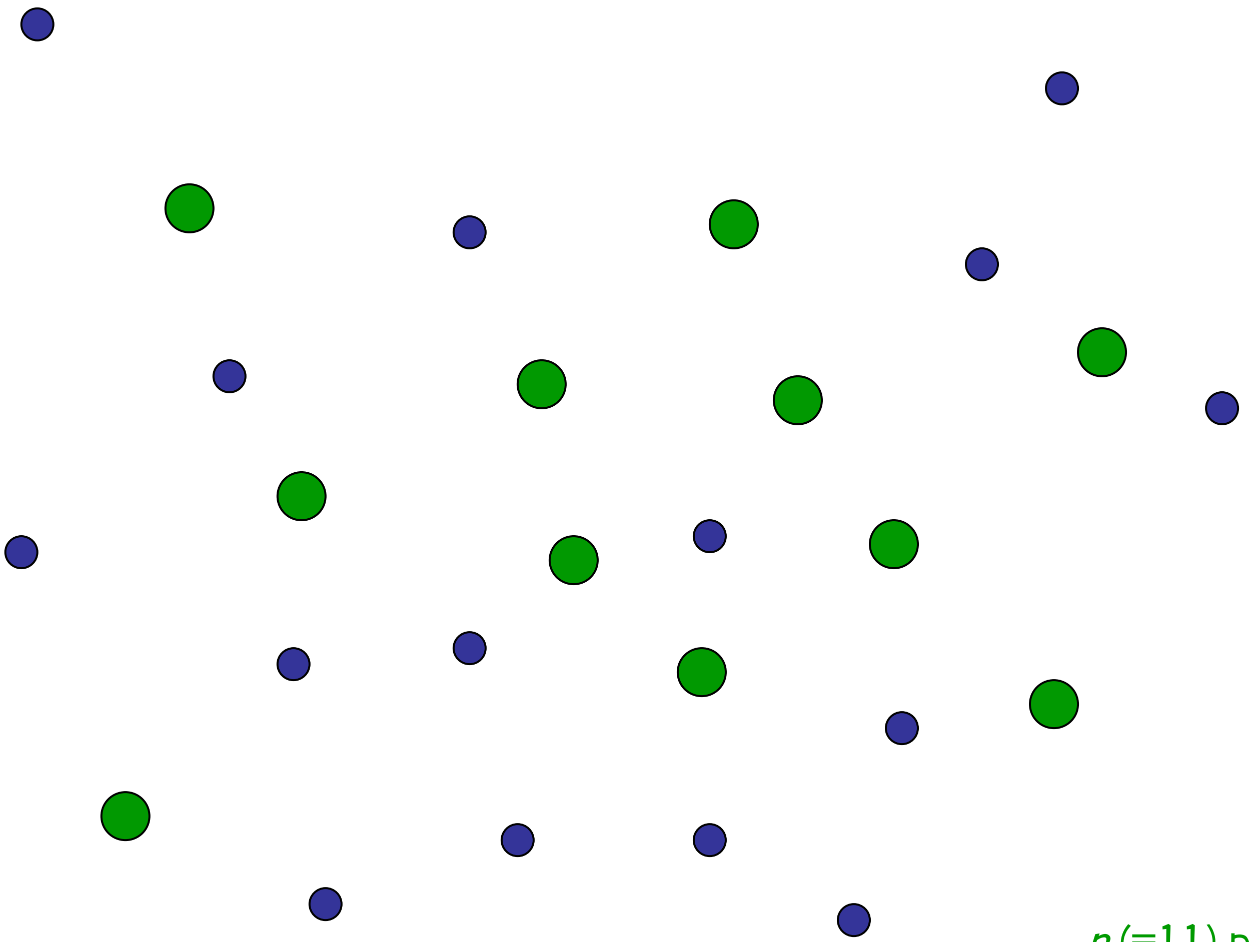
Joint work with
Renato Werneck, Princeton U.

Talk given at COPPE/UFRJ on August 8, 2003

Summary

- The p-median problem
- New swap-based local search
- GRASP
- Path-relinking
- GRASP with path-relinking using the new swap-based local search

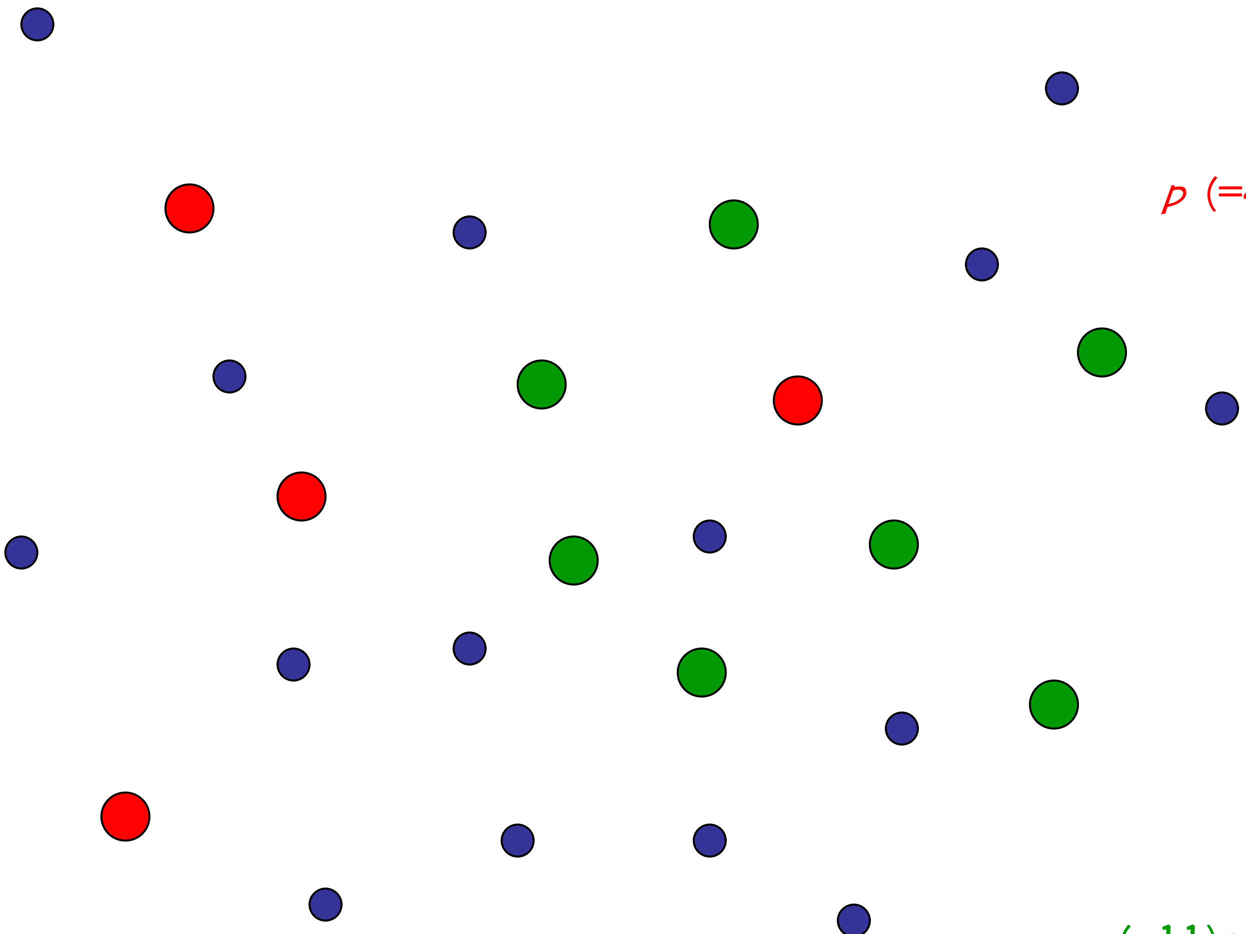
p-median problem



n (=11) potential service locations

m (=15) customers

p-median problem

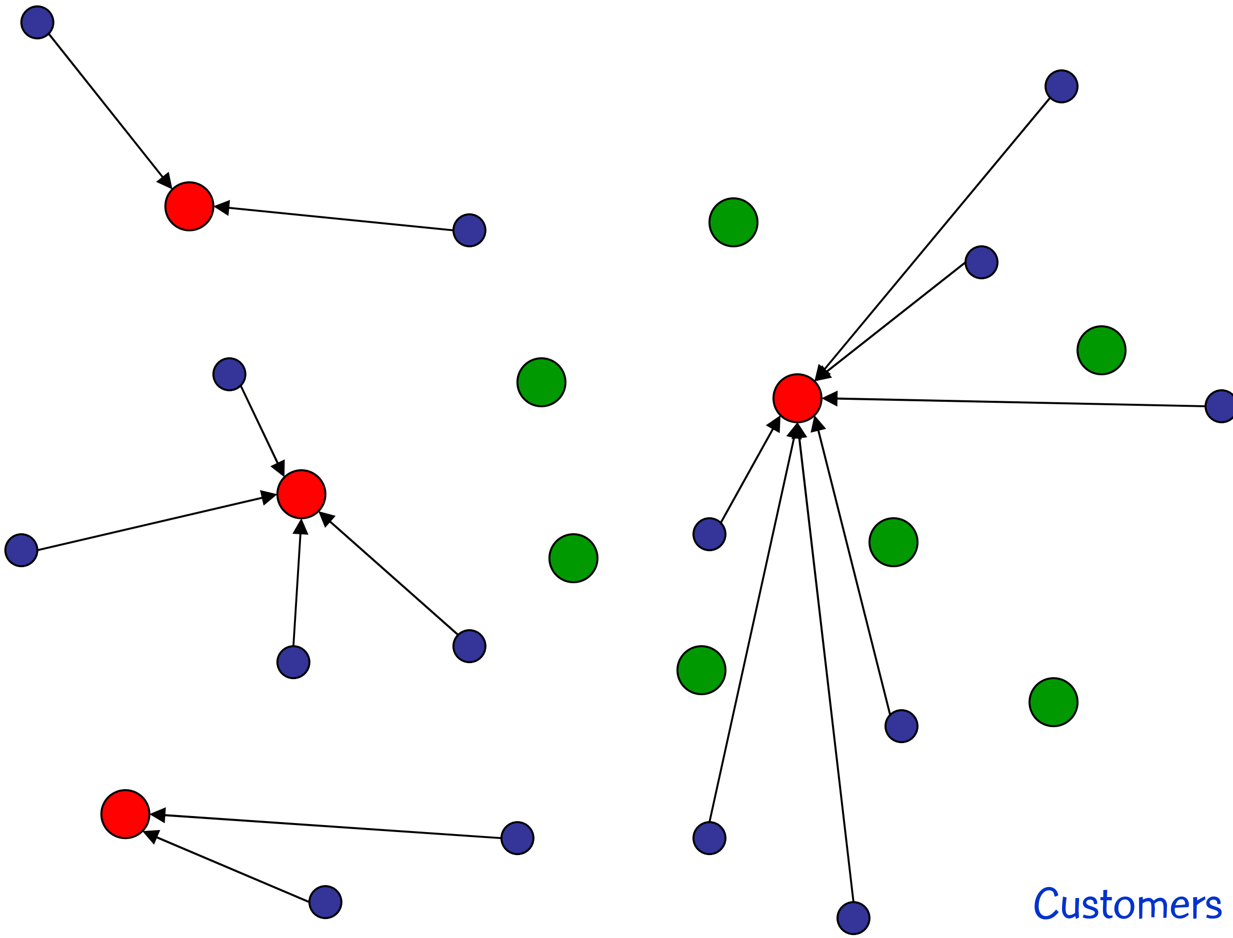


p (=4) service sites to be deployed

n (=11) potential service locations

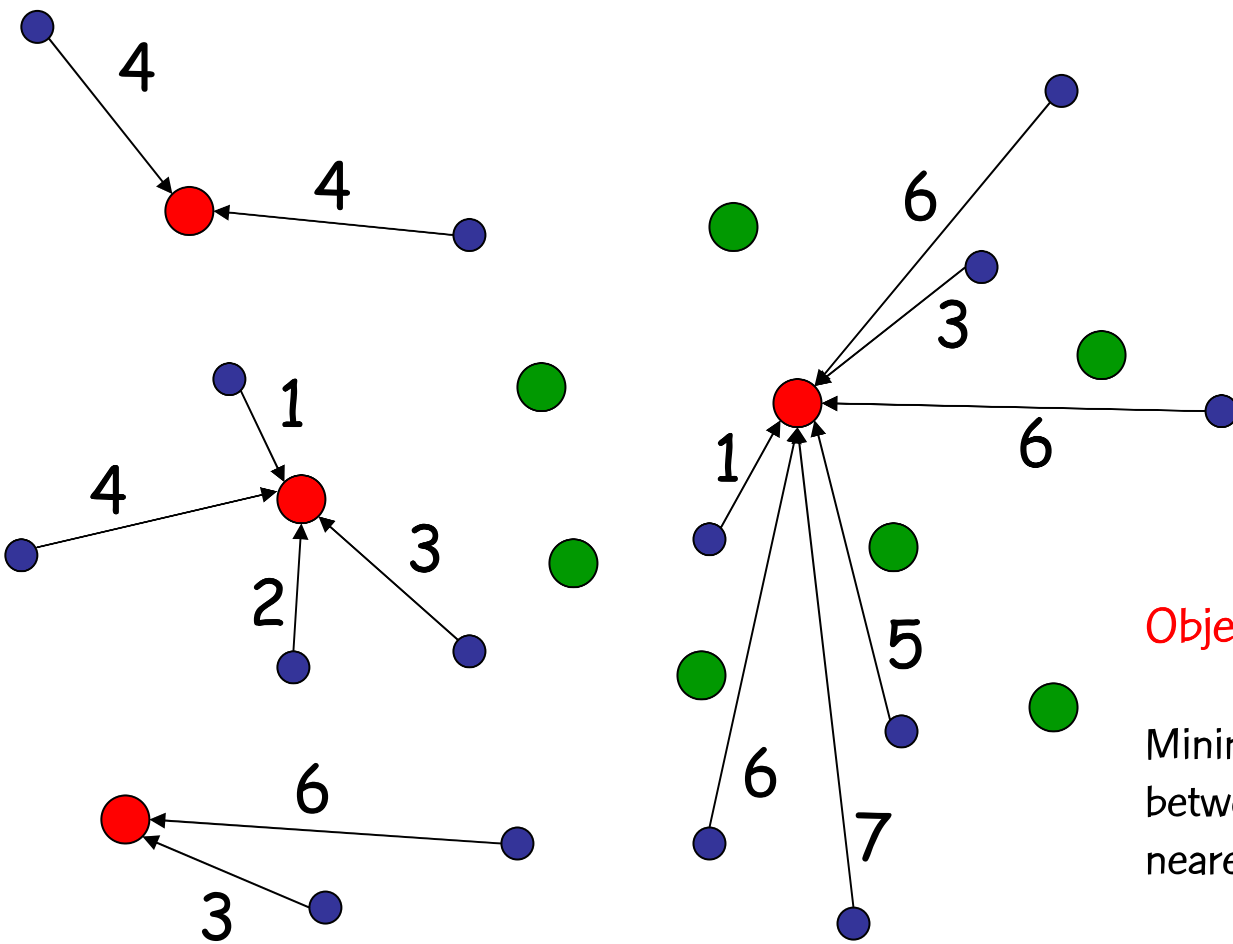
m (=15) customers

p-median problem



Customers home into nearest service center.

p-median problem

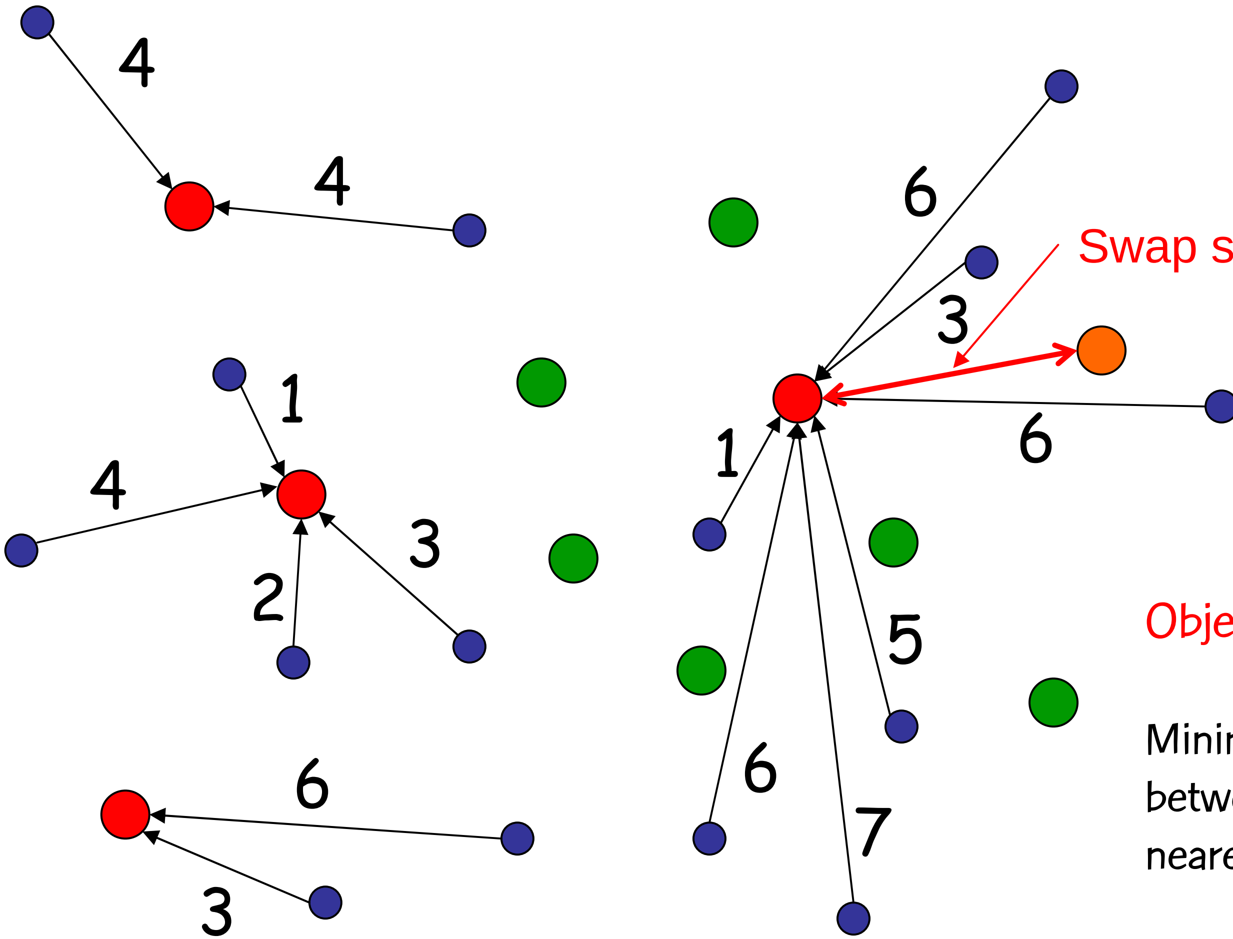


Objective of optimization:

Minimize sum of the distances between customers and their nearest service center.

Total distance = 61

p-median problem

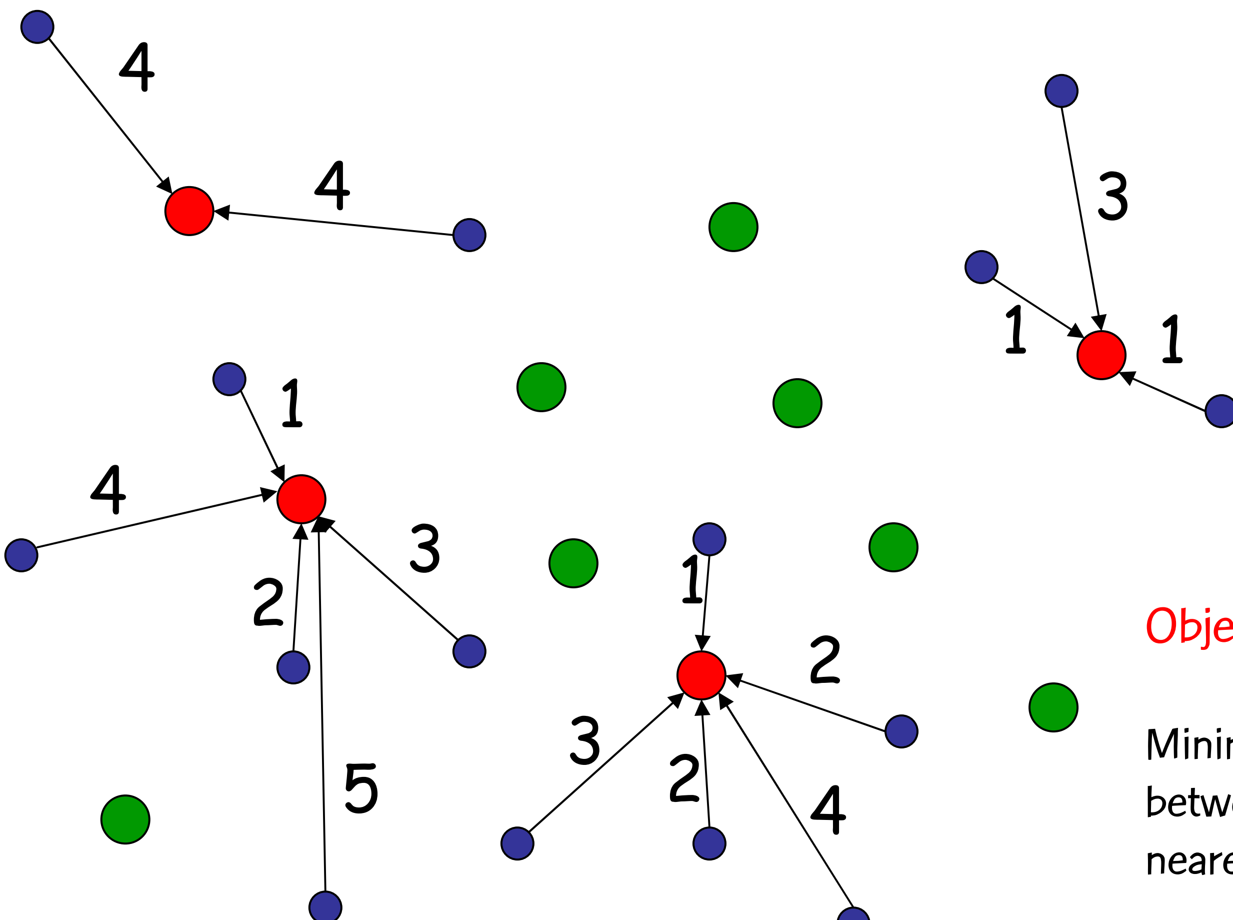


Objective of optimization:

Minimize sum of the distances between customers and their nearest service center.

Total distance = 61

p-median problem



Objective of optimization:

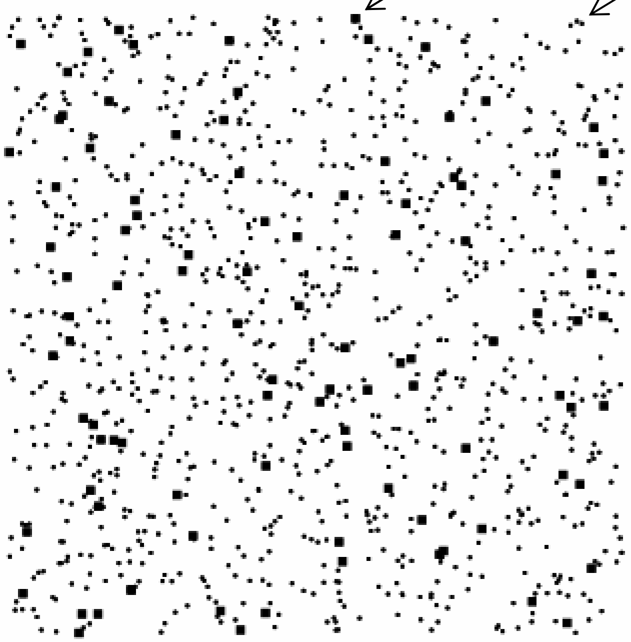
Minimize sum of the distances between customers and their nearest service center.

Total distance = 40 < 61

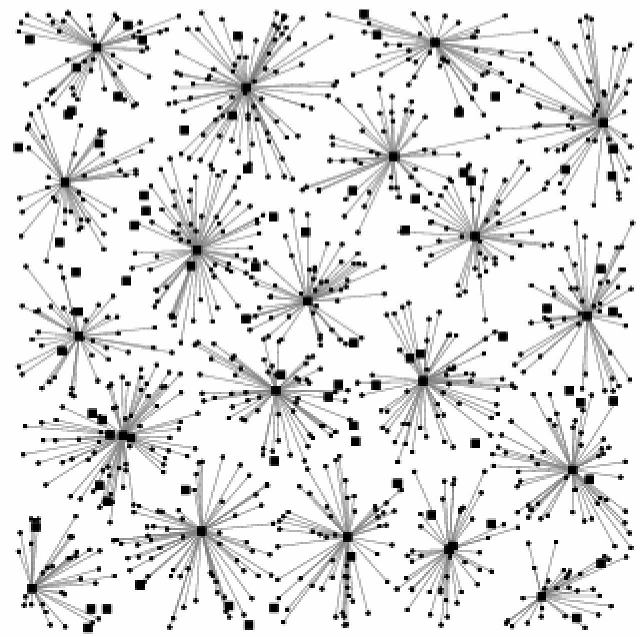
Example: 1000 customer locations, choose best 20 of 100 service locations

Potential service location (■)

Customer location (•)



Instance



Solution

The p-median problem

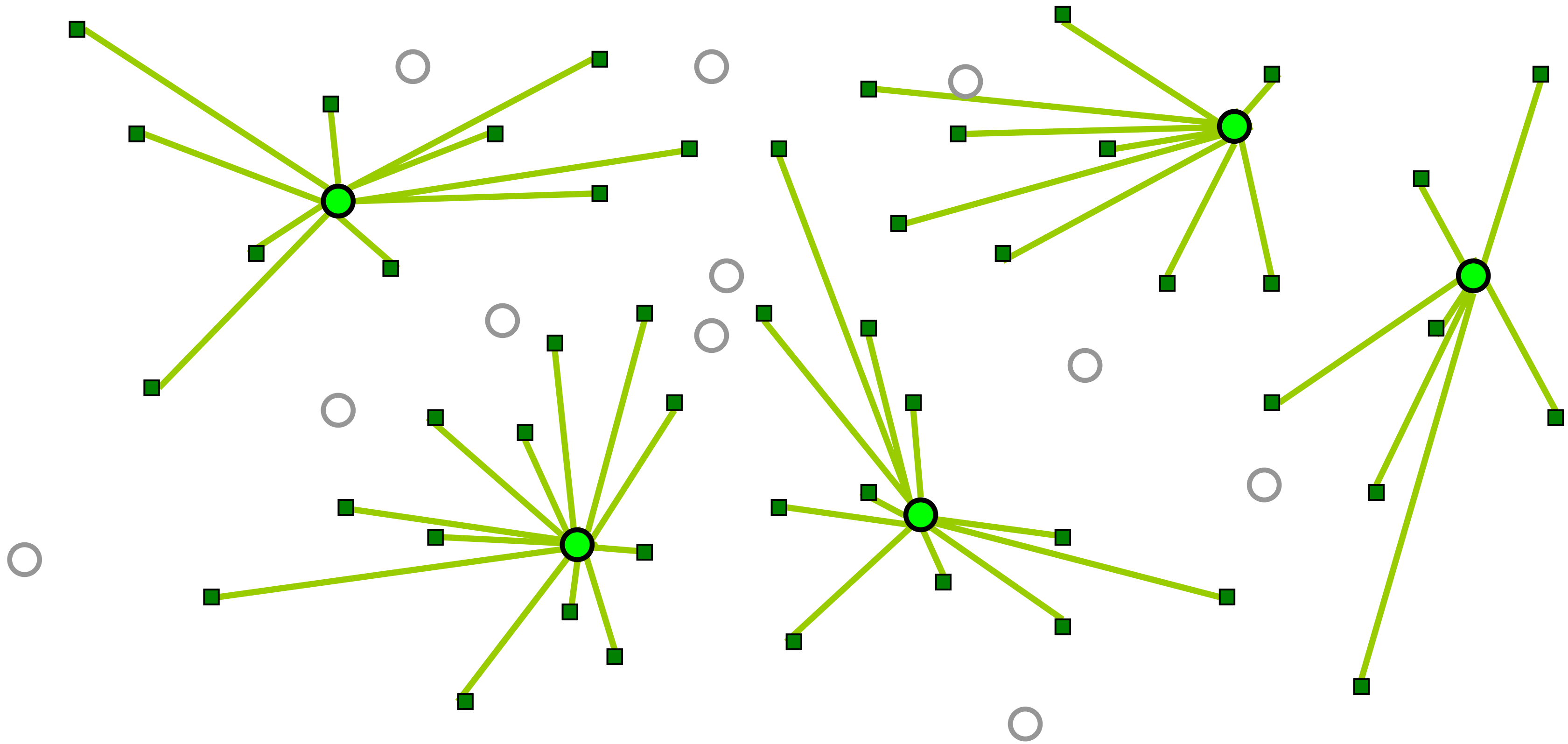
- Also known as the *k*-median problem.
- NP-hard (Kariv & Hakimi, 1979)
- Input:
  - a set U of n *users* (or *customers*);
  - a set F of m *potential facilities;*
  - a distance function (d: $U \times F \rightarrow \Re$);
  - the number of facilities p to open ($0 < p < m$).
- Output:
  - a set $S \subseteq F$ with p open facilities.
- Goal:
  - minimize the sum of the distances from each user to the closest open facility.

Swap-based local search

Basic Steps:

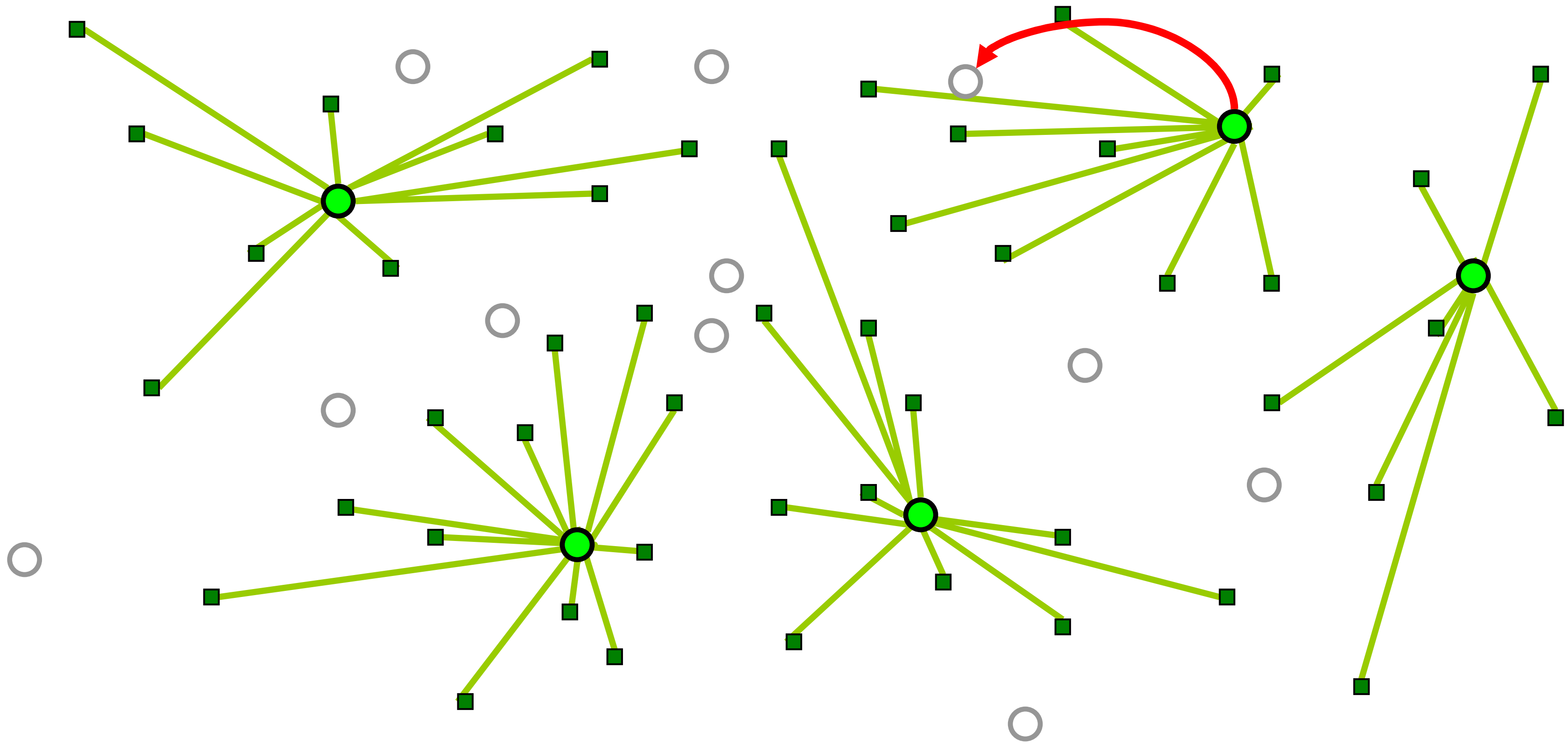
1. Start with some valid solution.
2. Look for a pair of facilities (f_i, f_r) such that:
   - f_i does not belong to the solution;
   - f_r belongs to the solution;
   - swapping f_i and f_r improves the solution.
3. If (2) is successful, swap f_i and f_r and repeat (2); else stop (a local minimum was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

Swap-based local search



original solution

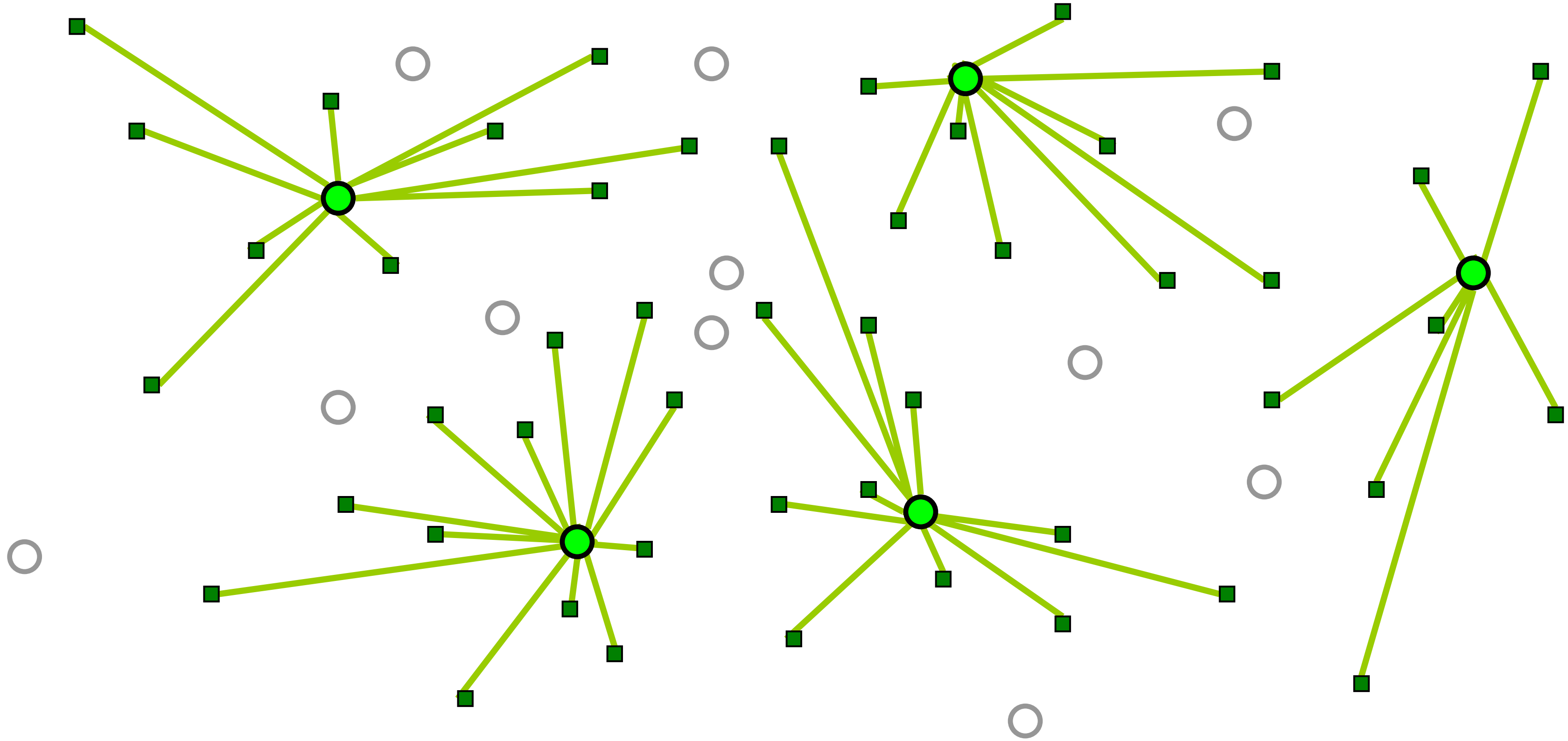
Swap-based local search



original solution

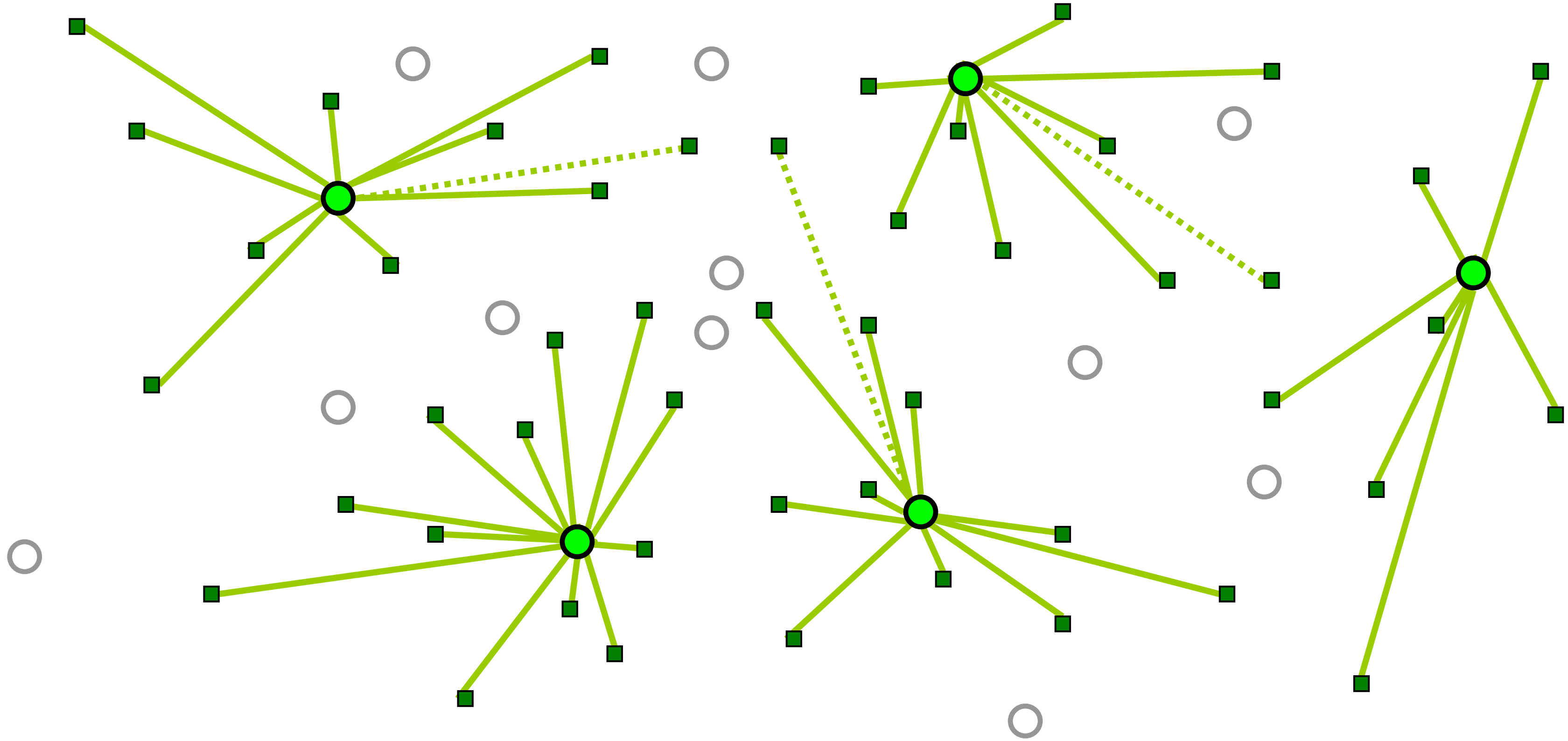
(not a local optimum)

Swap-based local search



improved solution

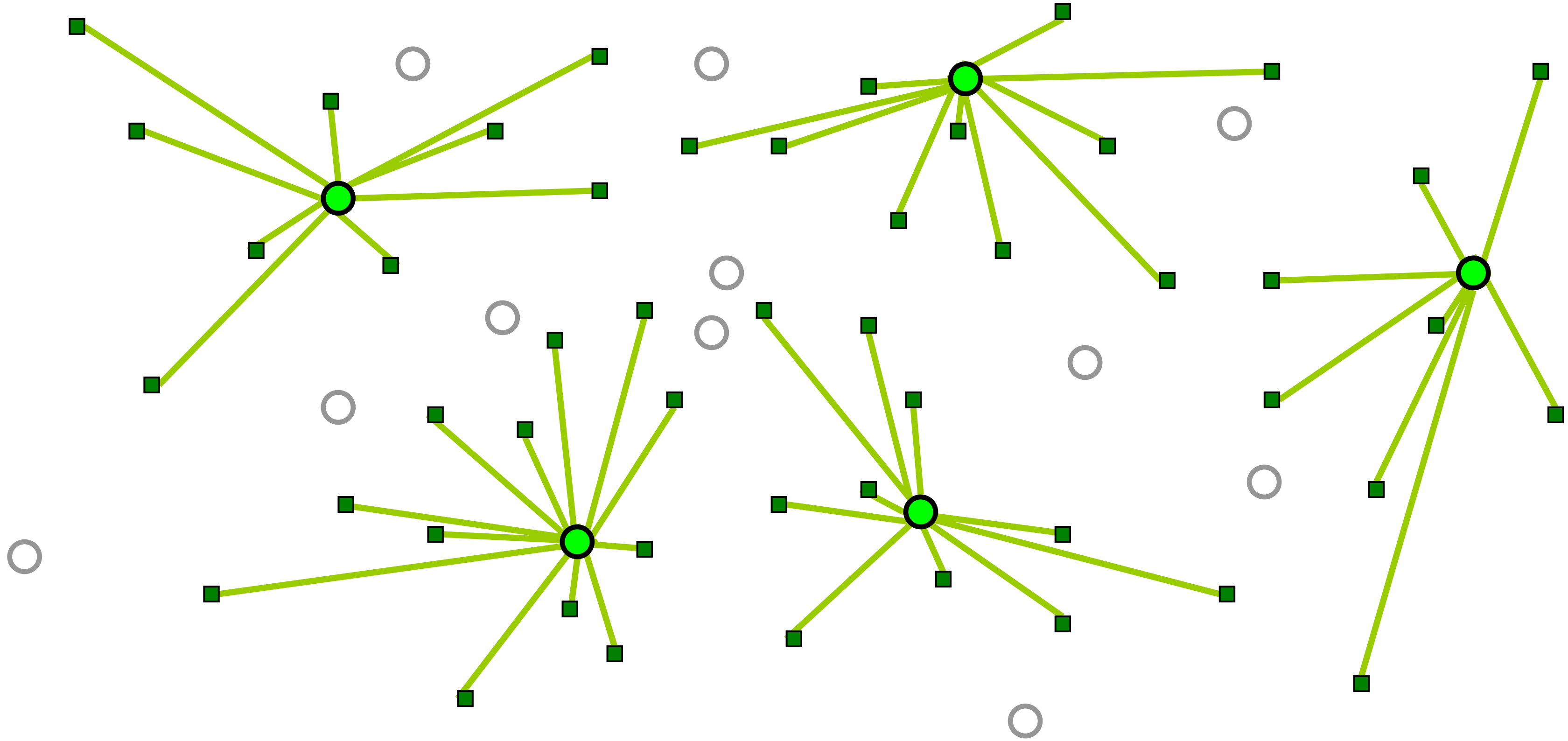
Swap-based local search



improved solution

(with wrong assignments)

Swap-based local search



improved solution

(with proper assignments)

Swap-based 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: 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 fact:
    - iterations are independent.
  - We use extra memory to avoid repeating previously executed calculations.

AT&T

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

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

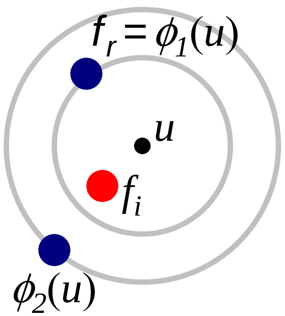
    (for a properly defined extra function).
  - extra can be thought of as a correction factor.

Correction factor

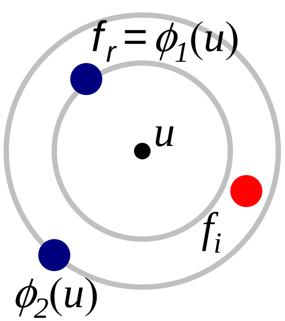
Things will go wrong for a user u iff:

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 and there will be no loss.
   - Loss overestimated by $[d(u, \phi_2(u)) - d(u, f_r)]$.
2. The new facility is farther from u than $\phi_1(u)$ is, 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 $\Rightarrow$ 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

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

extra is a matrix

This can be computed in $O(mn)$ time for all pairs.

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:

The contribution of each user can be computed independently.

Our implementation

function updateStructures $(S, u, loss, gain, extra, \phi_1, \phi_2)$

$f_r = \phi_1(u)$;

$loss[f_r]$ += $d(u, \phi_2(u)) - d(u, \phi_1(u))$;

forall $(f_i \notin S)$ **do** {

if $(d(u, f_i) < d(u, \phi_2(u)))$ **then**

$gain[f_i]$ += $\max\{0,\ d(u, \phi_1(u)) - d(u, f_i)\}$;

$extra[f_i, f_r]$ += $d(u, \phi_2(u)) - \max\{d(u, f_i), d(u, f_r)\}$;

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:
  - Determine closeness information: $O(pm)$ time
  - Compute gain, loss, and extra: $O(mn)$ time
  - Use gain, loss, and extra to find best swap: $O(pm)$ time
- That's the same complexity 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
```

AT&T

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, ϕ_1, ϕ_2)
  $\boxed{A := U;}$
  resetStructures$(gain, loss, extra)$;
  while (TRUE) **do** {
    forall $(u \in A)$ **do** updateStructures $(S, u, gain, loss, extra, \phi_1, \phi_2)$;
    $(f_r, f_i, profit)$:= findBestNeighbor $(gain, loss, extra)$;
    if $(profit \leq 0)$ **then break;**
    $A := \emptyset$;
    forall $(u \in U)$ **do**
      if $((\phi_1(u) = f_r)$ **or** $(\phi_2(u) = f_r)$ **or** $(d(u, f_i) < d(u, \phi_2(u))))$ **then**
        $A := A \cup \{u\}$;
      endif;
    endforall
    forall $(u \in A)$ **do**
    undoUpdateStructures$(S, u, gain, loss, extra, \phi_1, \phi_2)$;
    insert(S, f_i);
    remove(S, f_r);
    updateClosest$(S, f_i, f_r, \phi_1, \phi_2)$;
  endwhile
end localSearch

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

Our implementation

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

```
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

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 that 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   (f_r,f_i,profit) := findBestNeighbor (gain,loss,extra);
    if (profit ≤ 0) then break;
    A := ∅;
    forall (u∈U) do
      if ((φ1(u)=f_r) or (φ2(u)=f_r) or (d(u,f_i)<d(u,φ2(u))) then
        A := A∪{u};
      endif;
    endforall
    forall (u∈A) do
3   undoUpdateStructures(S,u,gain,loss,extra,φ1,φ2);
    insert(S,f_i);
    remove(S,f_r);
1   updateClosest(S,f_i,f_r,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 for each user: O (n) 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 per user.
- The second case could be a bottleneck, but in practice only a few users fall into this case.
  - Only these need to be tested.
  - This was observed by Hansen and Mladenović (1997).

AT&T

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 (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 as for straightforward approach)
- 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: Update 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 – Update 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 – Update 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: Update Structures

- Preprocessing step: Time
  - $O(nm \log m)$;
  - preprocessing step executed only once, even if local search is run several times.
- Preprocessing step: 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, the sets of users and potential facilities are the same.

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 than FI 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 (like GRASP, tabu search, VNS, ...).

AT&T

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

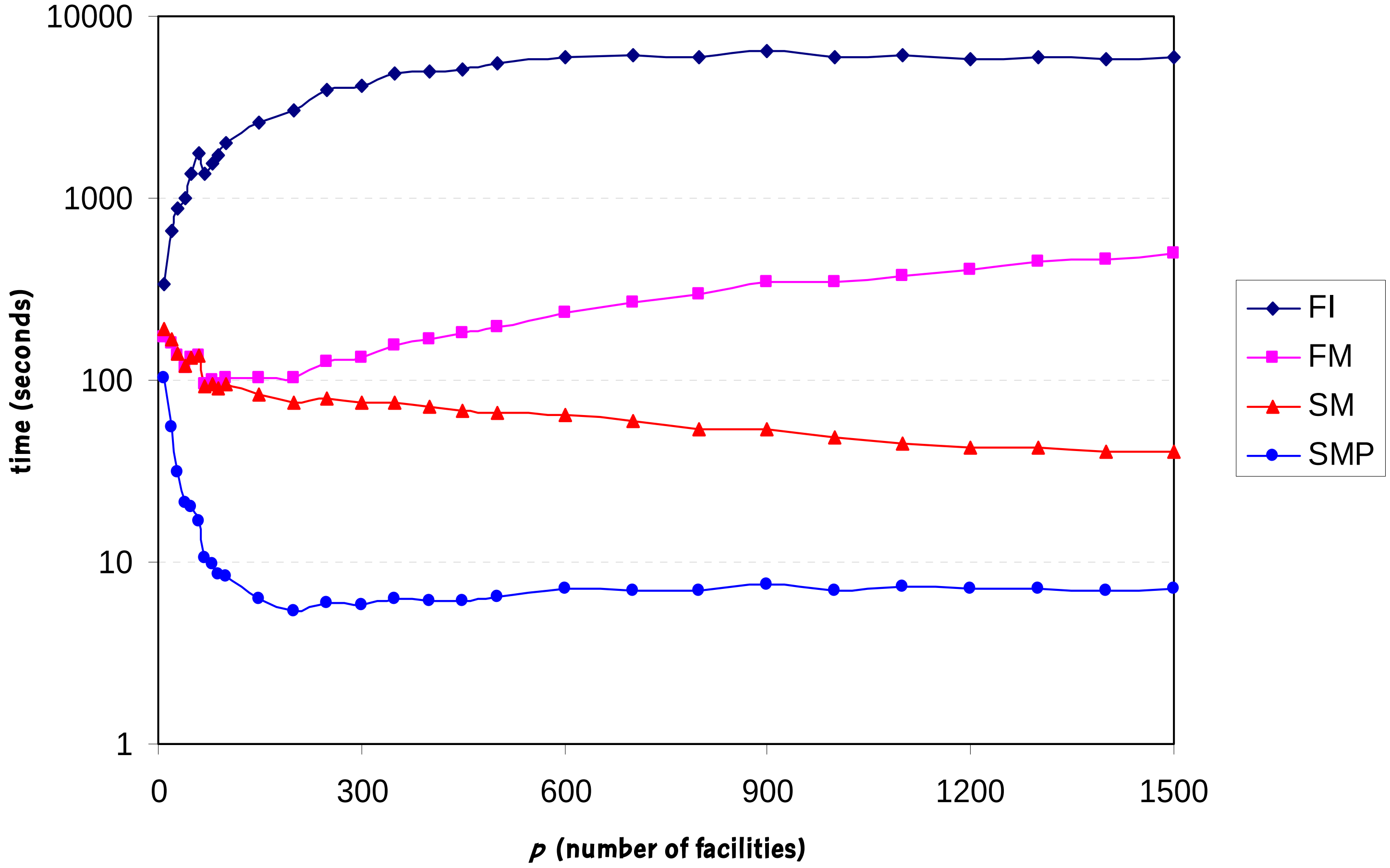
Speedups w.r.t. Whitaker's **FI** (worst cases):

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.

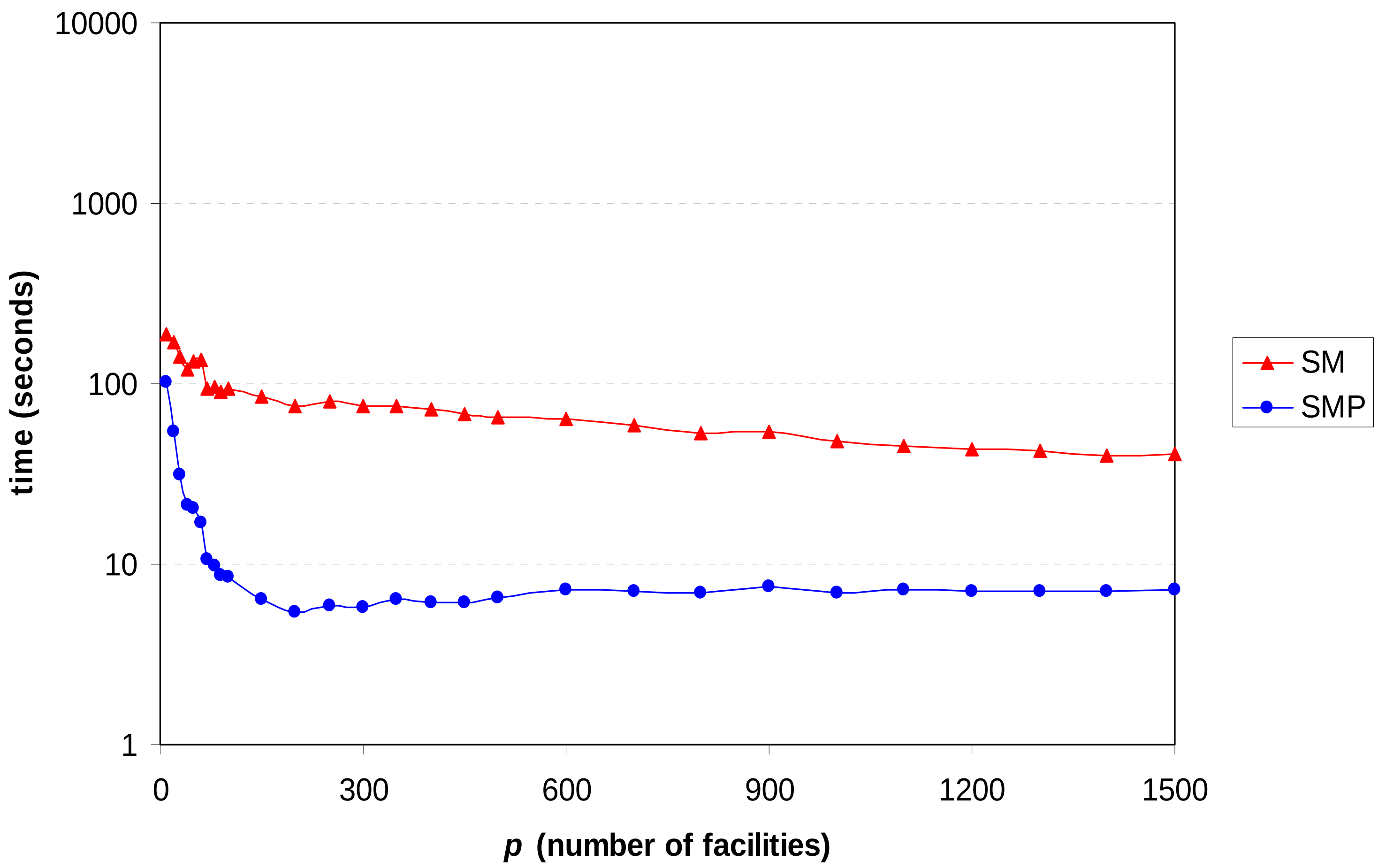
Results



Largest instance tested: 5934 users, Euclidean.

(preprocessing times not considered)

Results

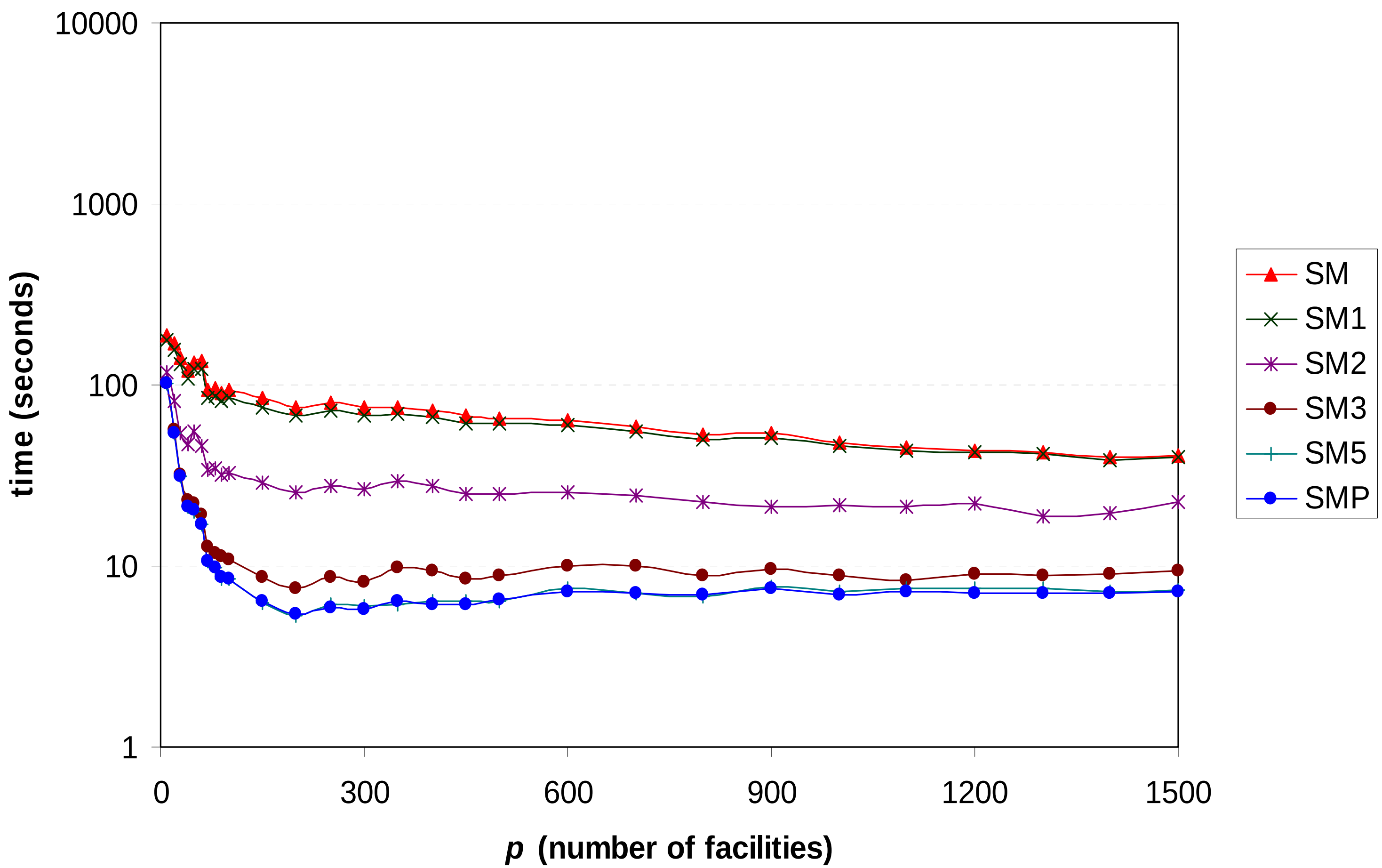


Note that preprocessing significantly accelerates the algorithm.

Results

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

Results



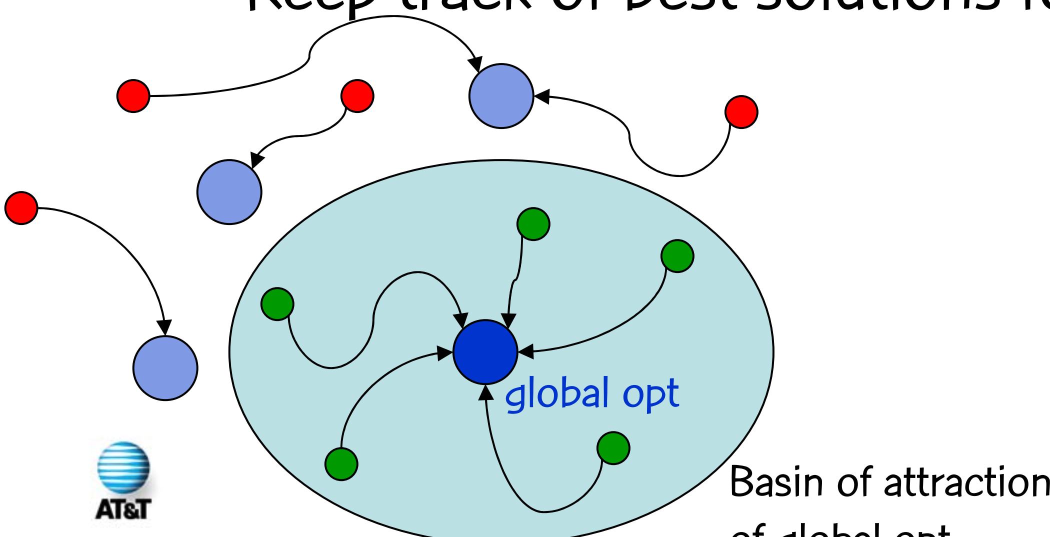
For this instance, $q = 5$ is already
as fast as the full version.

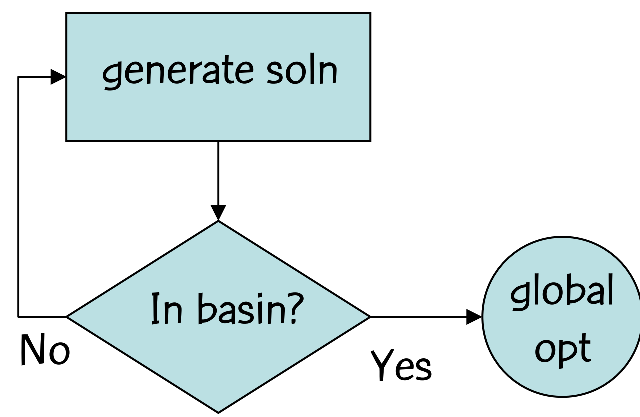
Final remarks on local search

- 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.

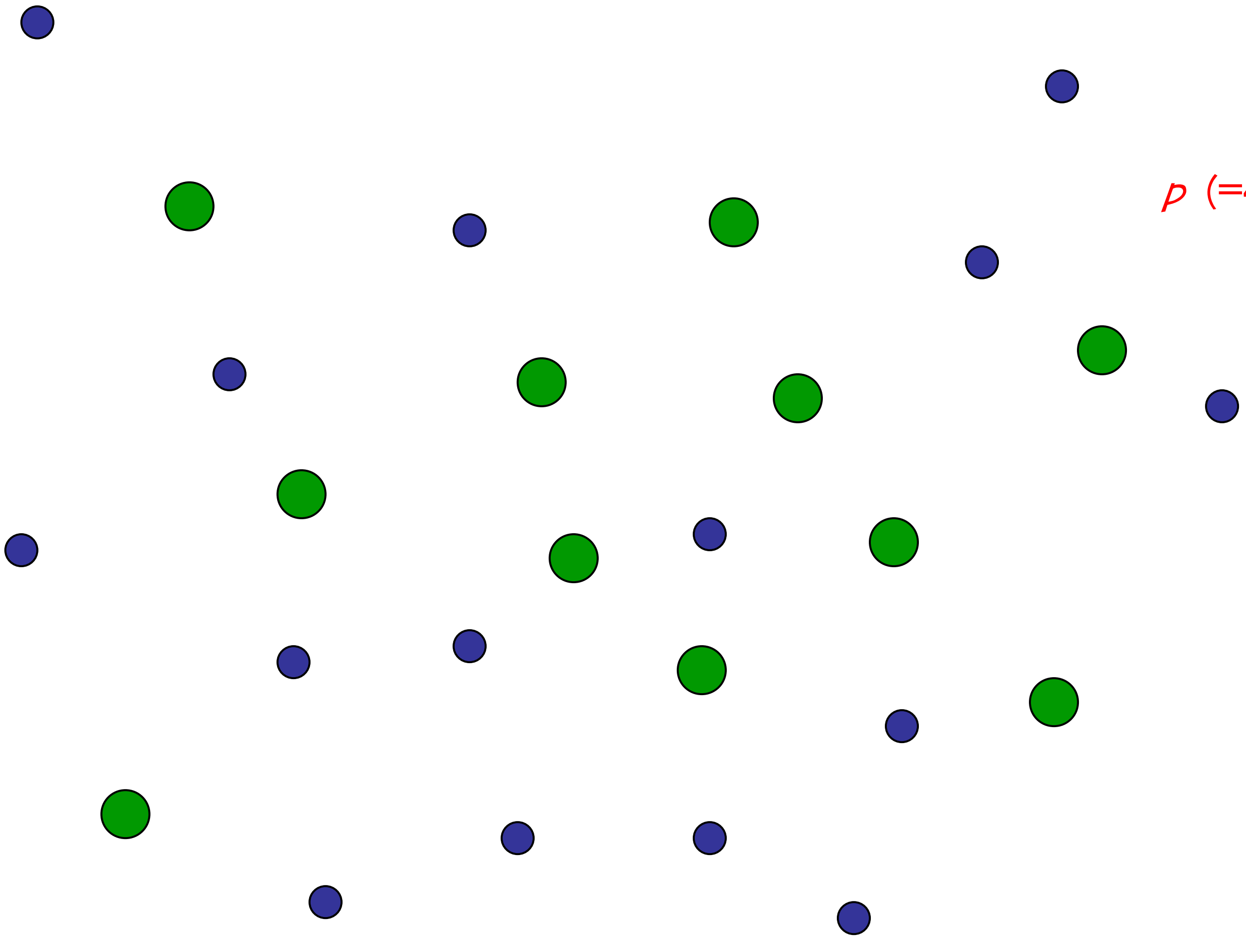
GRASP: greedy randomized adaptive search procedure

- Multi-start metaheuristic (Feo & Resende, 1989)
- Repeat:
  - Construct greedy randomized solution
  - Use local search to improve constructed solution
  - Keep track of best solutions found





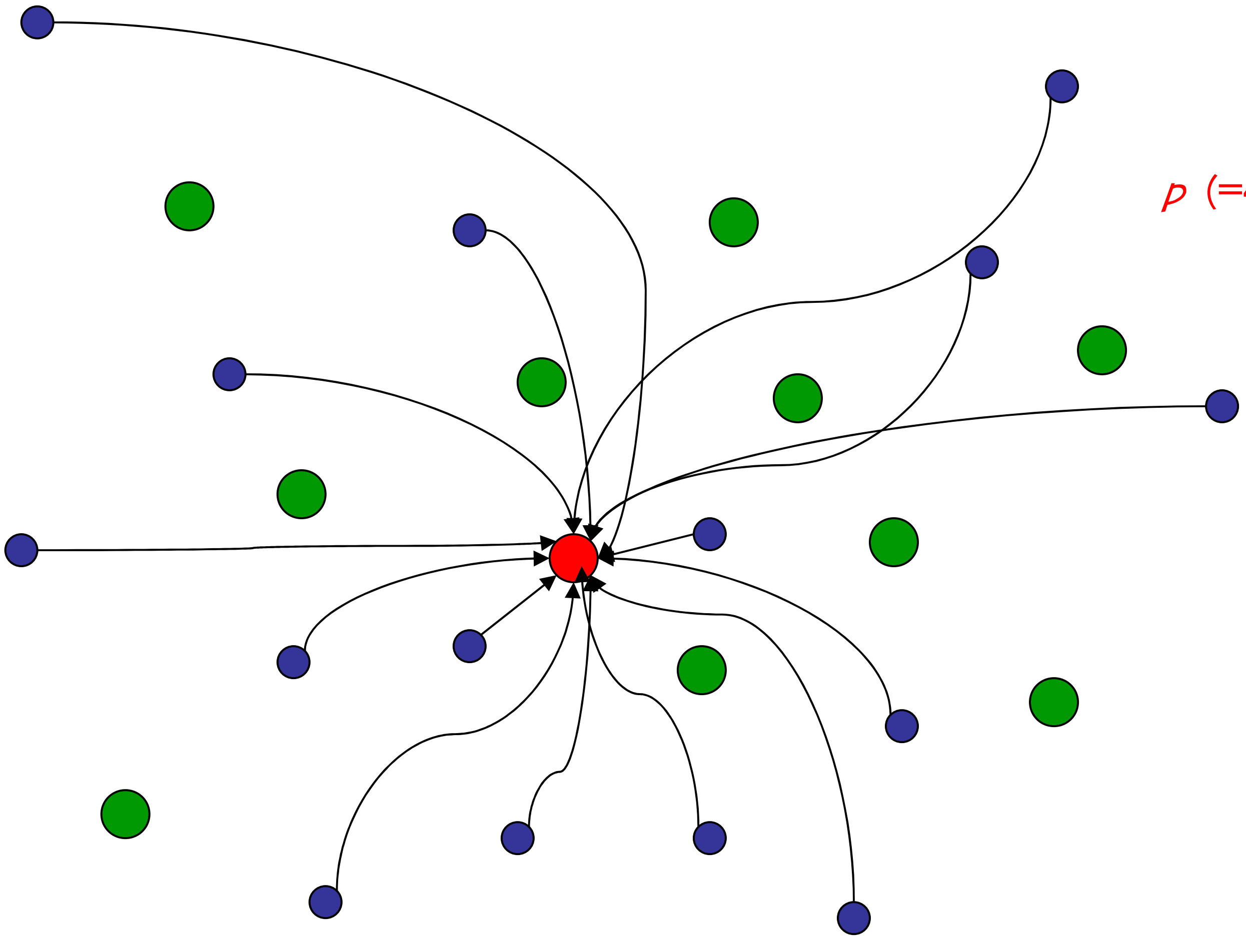
Greedy construction for p-median



p (=4) service sites to be deployed

Start with empty solution

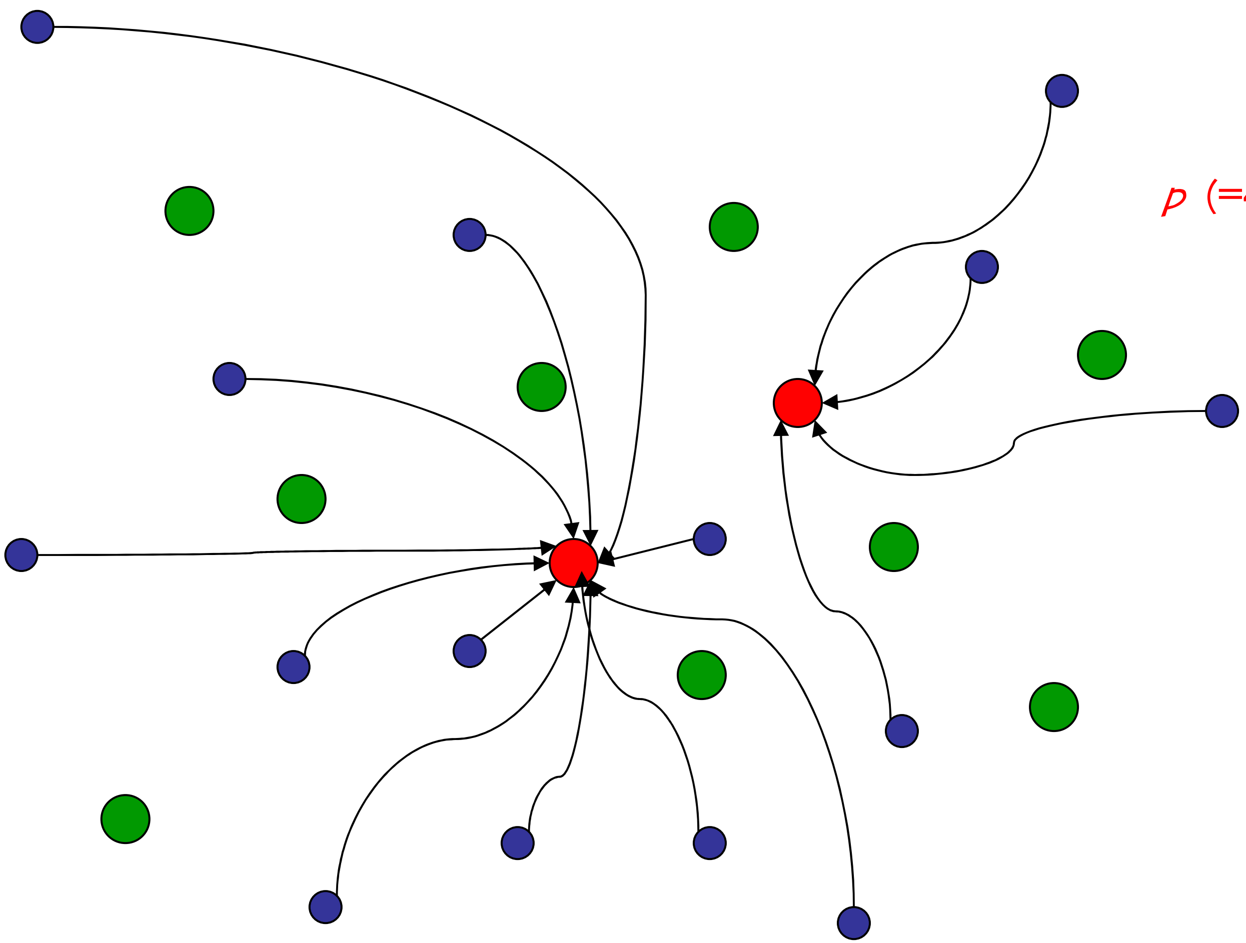
Greedy construction for p-median



p (=4) service sites to be deployed

Add most profitable center (least cost)

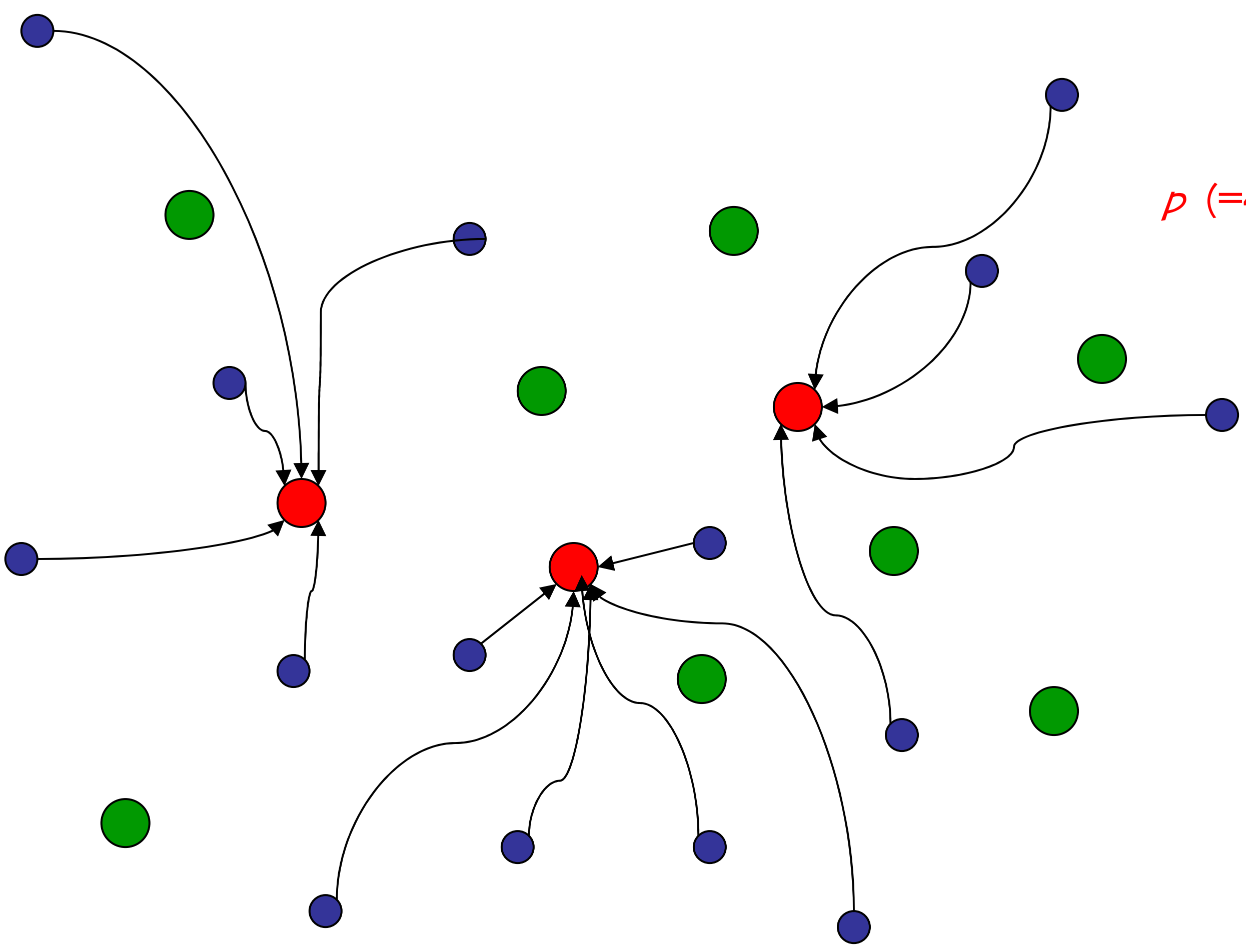
Greedy construction for p-median



p (=4) service sites to be deployed

Add most profitable center (One whose addition causes greatest drop in cost)

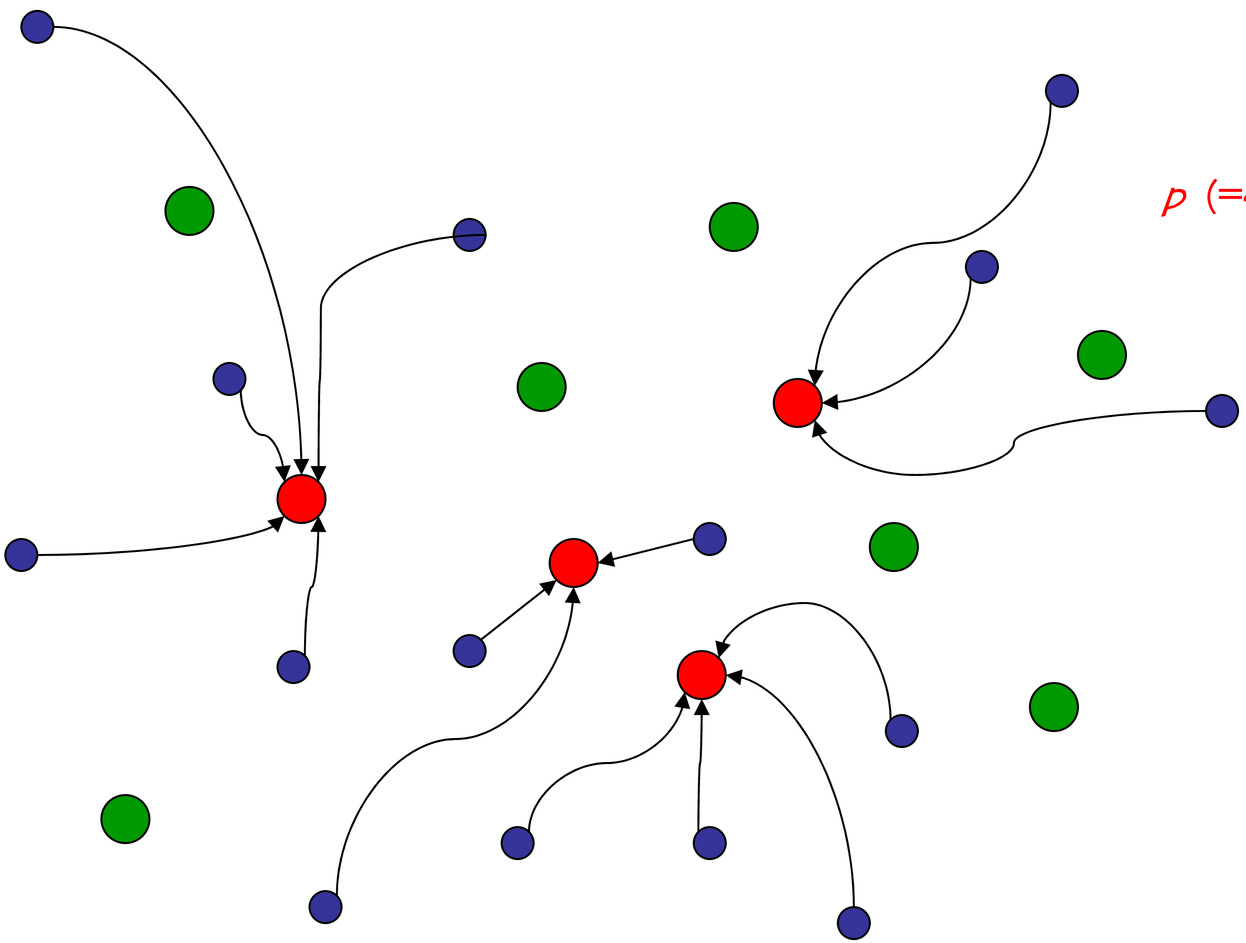
Greedy construction for p-median



p (=4) service sites to be deployed

Add most profitable center (One whose addition causes greatest drop in cost)

Greedy construction for p-median



p (=4) service sites to be deployed

Add most profitable center (One whose addition causes greatest drop in cost)

Randomized greedy

- Greedy construction cannot be used within GRASP framework:
  - Being deterministic, it yields identical solutions in all iterations

Randomized greedy

- Randomization needs to be added to greedy construction:
  - Random: select p sites at random ($O(m + pn)$ time)
  - Random plus greedy: select a fraction α of the p facilities at random, then complete in a greedy fashion ($O(pmn)$ time if α is not too close to 1)
  - Randomized greedy: similar to greedy, but choose randomly from $\lceil \alpha (m - i + 1) \rceil$ best options, where $0 \leq \alpha \leq 1$ is an input parameter ($O(pnm)$ time)

Randomized greedy

- Randomization needs to be added to greedy construction:
  - Proportional greedy: for each facility f_i compute how much would be saved if f_i were added to solution. Let $s(f_i)$ be this amount. Pick facility at random with probability proportional to $s(f_i) - \min_k s(f_k)$ (O(pmn) time)
  - Proportional worst: (Taillard, 1998) First facility chosen at random. Others one at a time. For each customer, compute the difference between how much its current assignment costs and how much the best assignment would cost. Select customer at random proportional to this difference and open closest facility. (O(mn) time)

Randomized greedy

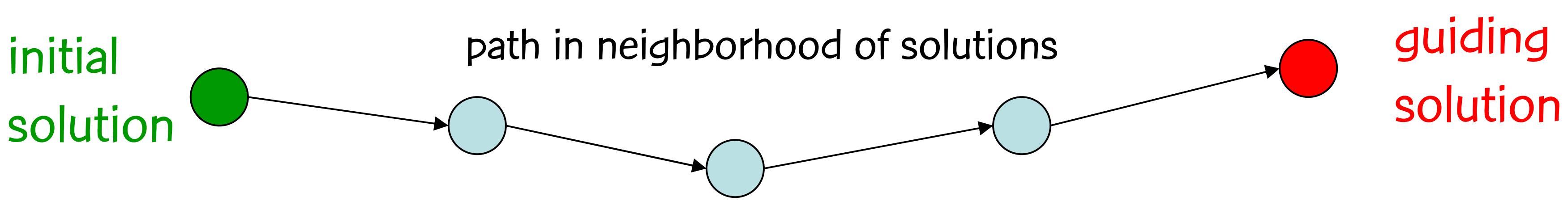
- After extensive testing, we chose this scheme:
  - Sample greedy: Similar to greedy. Instead of selecting among all possible options, consider only $q < m$ possible insertions (chosen uniformly at random). The most profitable facility is selected. Running time is $O(m+qpn)$. Idea is to make q small enough to reduce running time, while insuring a fair degree of randomization. We use $q = \lceil \log_2 (m/p) \rceil$.

Intensification

- Works with a pool of elite solutions.
- Occurs in two different stages:
  - Every GRASP iteration: newly generated GRASP solution is combined with an elite solution chosen from pool.
  - In post-optimization phase, solutions in the pool are combined themselves.
- Path-relinking is used to combine solutions.

Path-relinking (PR)

- Introduced in context of tabu and scatter search by Glover (1996, 2000):
  - Approach to integrate intensification & diversification in search.
- Consists in exploring trajectories that connect high quality solutions.



Path-relinking

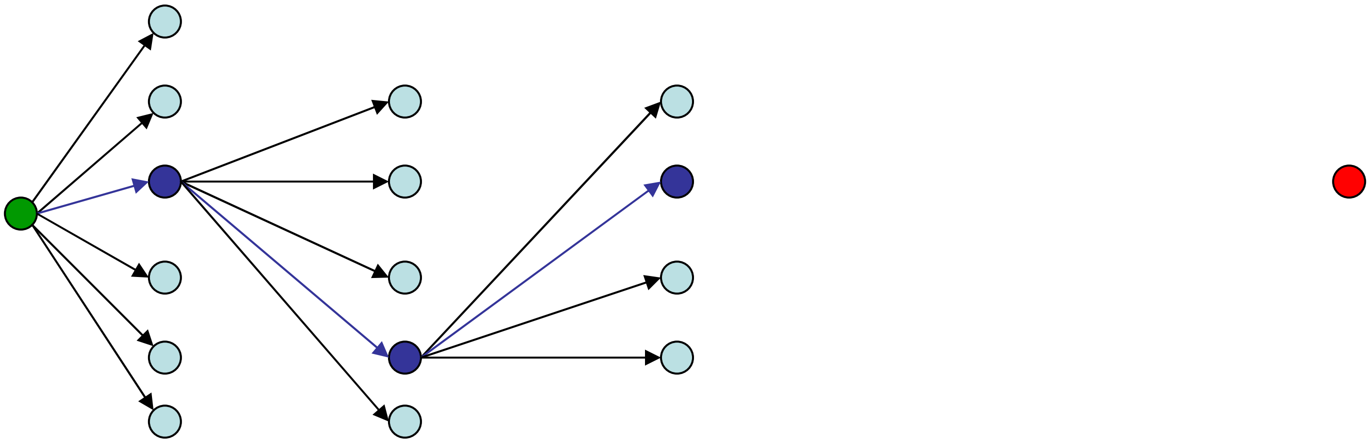
- Path is generated by selecting moves that introduce in the initial solution attributes of the guiding solution.
- At each step, all moves that incorporate attributes of the guiding solution are analyzed and best move is taken.

Path-relinking

- Path is generated by selecting moves that introduce in the initial solution attributes of the guiding solution.
- At each step, all moves that incorporate attributes of the guiding solution are analyzed and best move is taken.

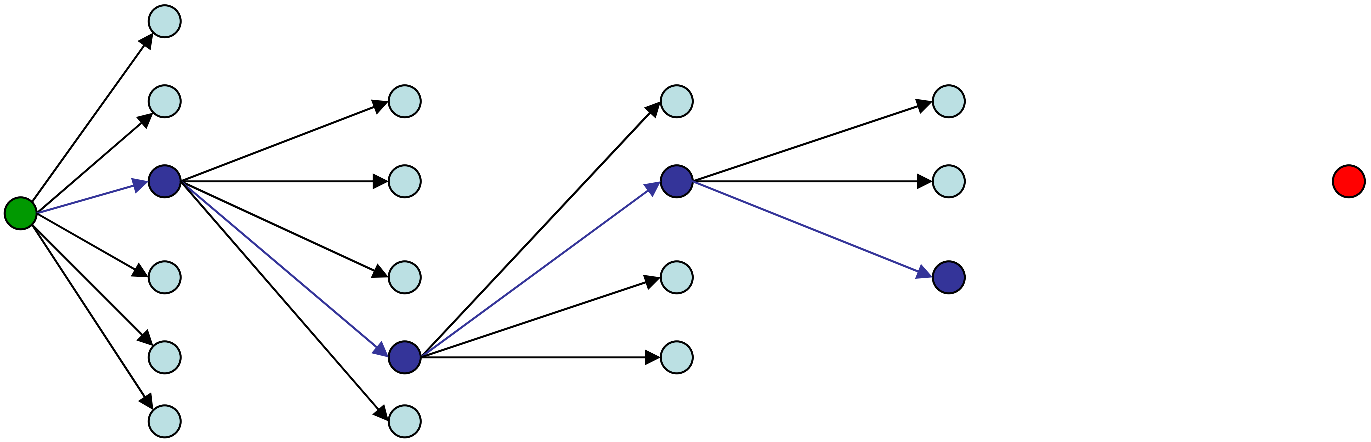
Path-relinking

- Path is generated by selecting moves that introduce in the initial solution attributes of the guiding solution.
- At each step, all moves that incorporate attributes of the guiding solution are analyzed and best move is taken.



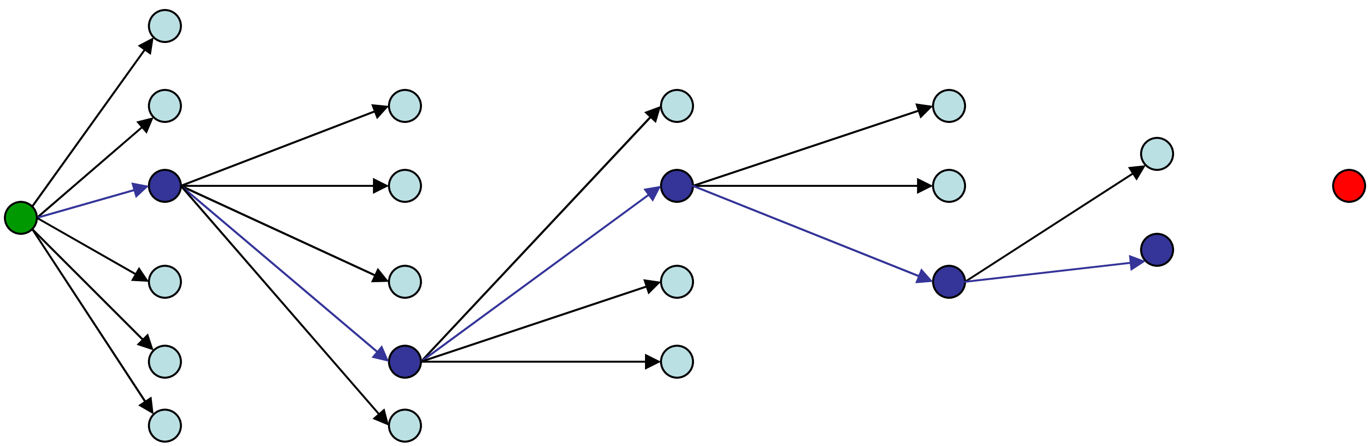
Path-relinking

- Path is generated by selecting moves that introduce in the initial solution attributes of the guiding solution.
- At each step, all moves that incorporate attributes of the guiding solution are analyzed and best move is taken.



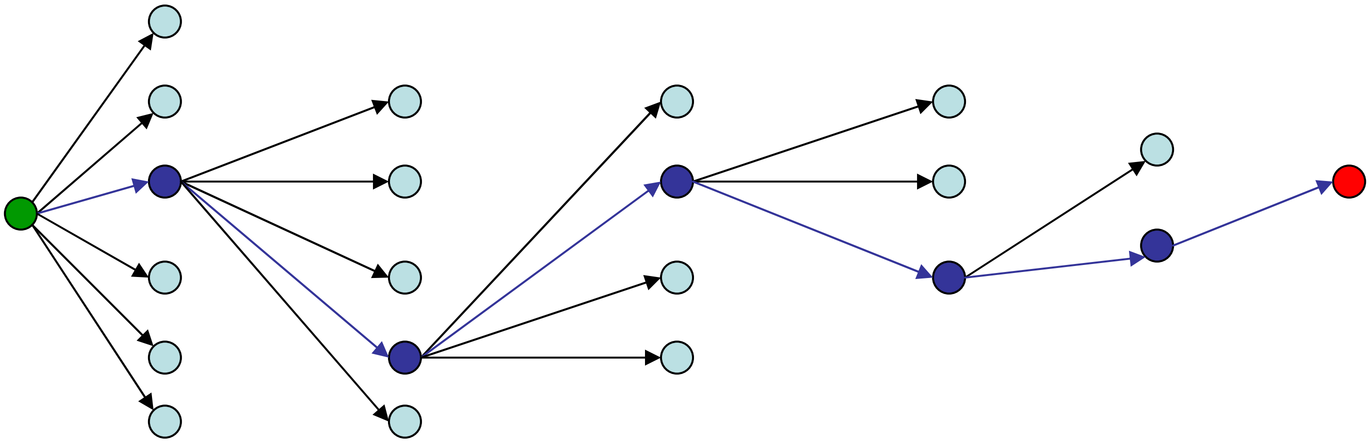
Path-relinking

- Path is generated by selecting moves that introduce in the initial solution attributes of the guiding solution.
- At each step, all moves that incorporate attributes of the guiding solution are analyzed and best move is taken.



Path-relinking

- Path is generated by selecting moves that introduce in the initial solution attributes of the guiding solution.
- At each step, all moves that incorporate attributes of the guiding solution are analyzed and best move is taken.



Output of PR usually is best solution in path.

Path-relinking

- Output of PR usually is best solution in path.
- We use a slight variation:
  - Outcome is best local minimum in path, where a solution in the path is a local minimum if it is both succeeded (immediately) and preceded (either immediately or through a series of same-valued solutions) in the path by strictly worse solutions.
  - If path has no local minimum, one of the path's extreme solutions is returned with equal probability.
  - When PR fails, our scheme tries to increase diversity by selecting some solution other than the extremes of the path.

Path-relinking

- We augment the intensification by performing a full local search on the solution produced by PR.
  - Usually much faster than local search on constructed solution since PR solutions are either a local minimum or very close to a local minimum.
  - A nice side effect of this is increased diversity, since we are free to use facilities that did not belong to either extreme solution of the path.

Path-relinking & local search

- Steps of path-relinking are very similar to the local search described earlier. Two main differences:
  - Number of allowed moves is restricted: only elements in symmetric difference $S_2 \setminus S_1$ can be inserted, and the ones in $S_1 \setminus S_2$ can be removed.
  - Non-improving moves are allowed.
- These differences are subtle enough to be easily incorporated into the basic implementation of the local search procedure (both procedures share the same code).

Pool management

- Relinking between a pair of similar solutions is less likely to be successful.
- Pool must support two essential operations:
  - Addition of new solutions;
  - Selection of a solution for path-relinking.

Pool management: Updates

- For a solution S with cost $v(S)$ to be added to the pool, two conditions must be met:
  - The symmetric difference between S and all solutions currently in the pool whose value is less than S must be at least 4.
    - Path relinking between solutions that differ in fewer than four facilities cannot produce solutions that are better than the extremes.
  - If the pool is full, S must be at least as good as the worst elite solution.

Pool management: Selection

- Previous work has selected an element from the pool, uniformly at random, to combine with *S*.
- However, this often results in selecting a solution that is too similar to *S*.
- We pick at random, but not uniformly. We use probabilities proportional to their symmetric difference with respect to *S*.
  - In paper, we show that this pays off.

Path relinking: Post-optimization

a) Start with pool found at end of GRASP: P_0 ;
Set $k = 0$;

b) Combine with path-relinking all pairs of solutions in pool P_k ;

c) Solutions obtained by combining solutions in P_k are added to a new pool P_{k+1} following same constraints for updates as before;

d) If best solution of P_{k+1} is better than best solution of P_k , then set $k = k + 1$, and go to step (b);

Results: Algorithmic setup

- Constructive procedure: sample greedy.
- Path-relinking is done during GRASP and as post-optimization.
- Path-relinking is performed from best to worst during GRASP, and from worst to best during post-optimization.
- Solutions are selected from pool during GRASP using biased scheme.
- GRASP iterations: 32
- Size of pool of elite solutions: 10

Results: Test problems

- TSP: Set of points on the plane (74 instances with 1400, 3038, and 5934 nodes)
  - 1400 node instance: $p = 10, 20, \ldots 450, 500$
  - 3038 node instance: $p = 10, 20, \ldots 950, 1000$
  - 5934 node instance: $p = 10, 20, \ldots 1400, 1500$
- ORLIB: From Beasley's ORLibrary (40 instances with 100 to 900 nodes and p from 5 to 200)
- SL: slight extension of ORLIB (3 instances with 700 nodes ($p = 233$), 800 nodes ($p = 267$), and 900 nodes ($p = 300$).

Results: Test problems

- GR: Galvão and ReVelle (1996) (16 instances with two graphs having 100 and 150 nodes and eight values of p between 5 and 50).
- RW: Resende & Werneck (2002) of completely random distance matrices. Distance between each facilty and customer is integer taken at random in interval $[1,n]$, where n is the number of customers. 28 instances with 100, 250, 500, and 1000 customers and different values of p.

Results: Compared with best known solutions

Instance	# Instances	# Ties	# Improved
TSP: fl1400	18	6	12
TSP: pcb3038	28	7	21
TSP: rl5934	28	9	19
ORLIB*	40	40	0
SL*	3	3	0
GR*	16	16	0

* Optimal solution known for all instances in ORLIB, SL, and GR.

Results: Other methods

- VNS: Variable neighborhood search by Hansen and Mladenović (1997)
- VNDS: Variable neighborhood decomposition search by Hansen, Mladenović, and Perez-Brito (2001)
- LOPT: Local optimization method by Taillard (1998)
- DEC: Decomposition procedure by Taillard (1998)
- LSH: Lagrangean-surrogate heuristic by Senne and Lorena (2000)
- CGLS: Column generation with Lagrangean/surrogate relaxation by Senne and Lorena (2002)

GRASP vs other methods

series	GRASP	CGLS	DEC	LOPT	LSH	VNDS	VNS
GR	0.009				0.727		
SL	0.000	0.691			0.332		
ORLIB	0.000	0.101			0.000	0.116	0.007
fl1400	0.031					0.071	0.191
pcb3038	0.025	0.043	4.120	0.712	2.316	0.117	0.354
rl5924	0.022					0.142	

Mean percentage deviation w.r.t best known solution.

Green is best algorithm; Red when not all instances tested; Black not tested.

AT&T

GRASP vs other methods

series	GRASP 196 MHz R10000	CGLS Sun Ultra 30	DEC 195 MHz R10000	LOPT 195 MHz R10000	LSH Sun Ultra 30	VNDS 147 MHz UltraSparc	VNS Sun SparcStation 10
GR	1.000				1.110		
SL	1.000	0.510			24.20		
ORLIB	1.000	55.98			4.130	0.460	5.470
fl1400	1.000					0.580	19.01
pcb3038	1.000	9.550	0.210	0.350	1.670	2.600	30.94
rl5924	1.000					2.930	

Mean ratio of running times w.r.t. GRASP.

Green GRASP is faster; Red GRASP is slower; Black not tested.

Concluding remarks

- New heuristic algorithm for p-median problem.
- We show that the method is remarkably robust:
  - Handles a wide variety of instances.
  - Obtains results competitive with those found by best heuristics in the literature.
- Our method is a valuable candidate for a general-purpose solver for the p-median problem.

Concluding remarks

- We do not claim our method is the best in every circumstance.
- Other methods are able to produce results of remarkably good quality, often at the expense of higher running times:
  - VNS (Hansen & Mladenović, 1997) is specially succesful for graph instances;
  - VNDS (Hansen, Mladenović, and Perez-Brito, 2001) is strong on Euclidean instances and very fast on problems with small p;
  - CGLS (Senne & Lorena, 2002) can obtain very good results for Euclidean instances and provides good lower bounds.

Concluding remarks

- Papers:
  - M.G.C. Resende and R. Werneck, "On the implementation of a swap-based local search procedure for the p-median problem," ALENEX03, 2003:

    http://www.research.att.com/~mgcr/doc/pmedianls.pdf
  - M.G.C. Resende and R. Werneck, "A GRASP with path-relinking for the p-median problem," submitted to J. of Heuristics (2002):

    http://www.research.att.com/~mgcr/doc/gpmedian.pdf
- Code: http://www.research.att.com/~mgcr/popstar
- Slides: http://www.research.att.com/~mgcr/talks/gpmedian.pdf