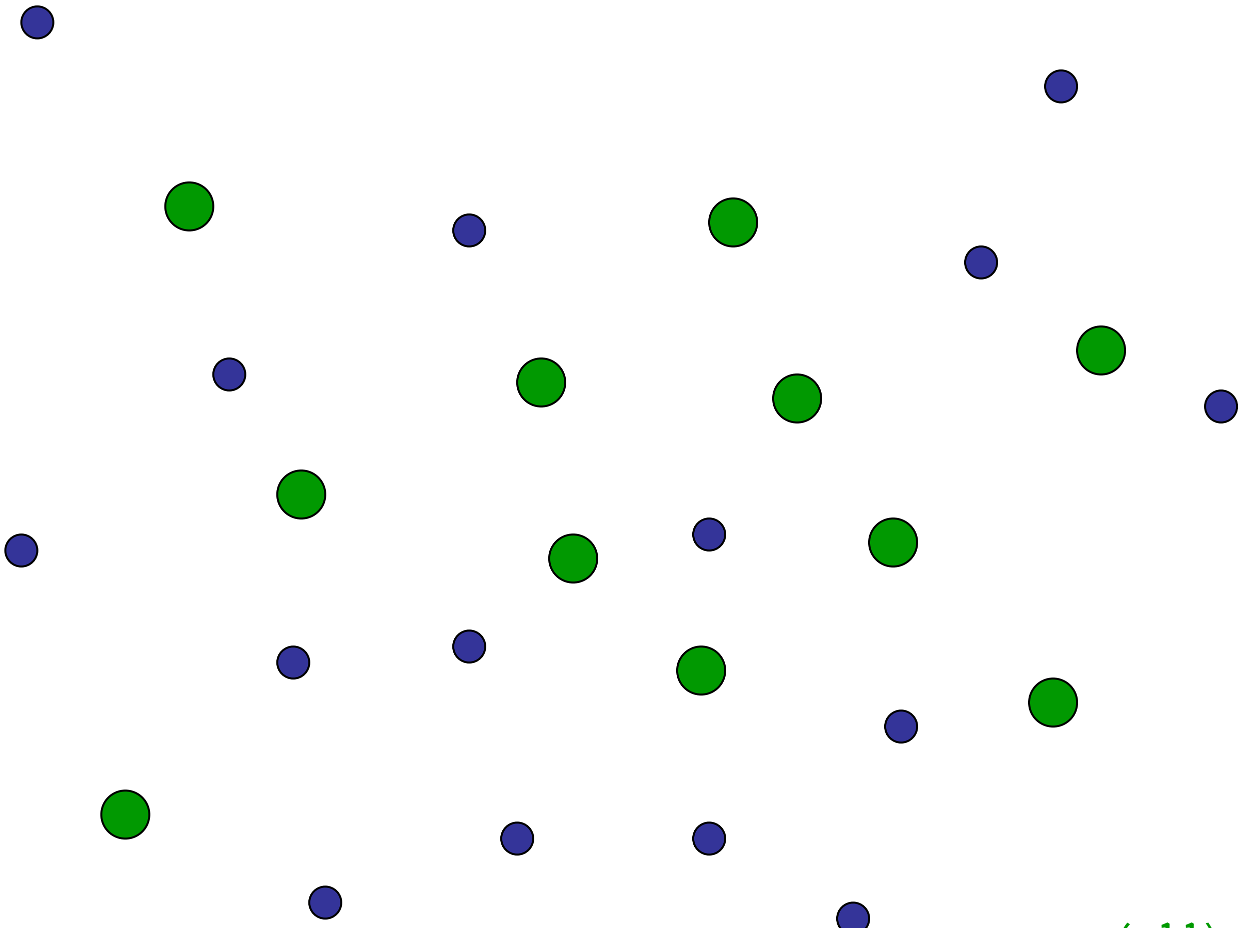
Mathematical Programming
in Rio
Búzios, November 9-12, 2003

A hybrid heuristic for the p-median problem

Maurício G.C. RESENDE
AT&T Labs Research
USA

Renato F. WERNECK
Princeton University
USA

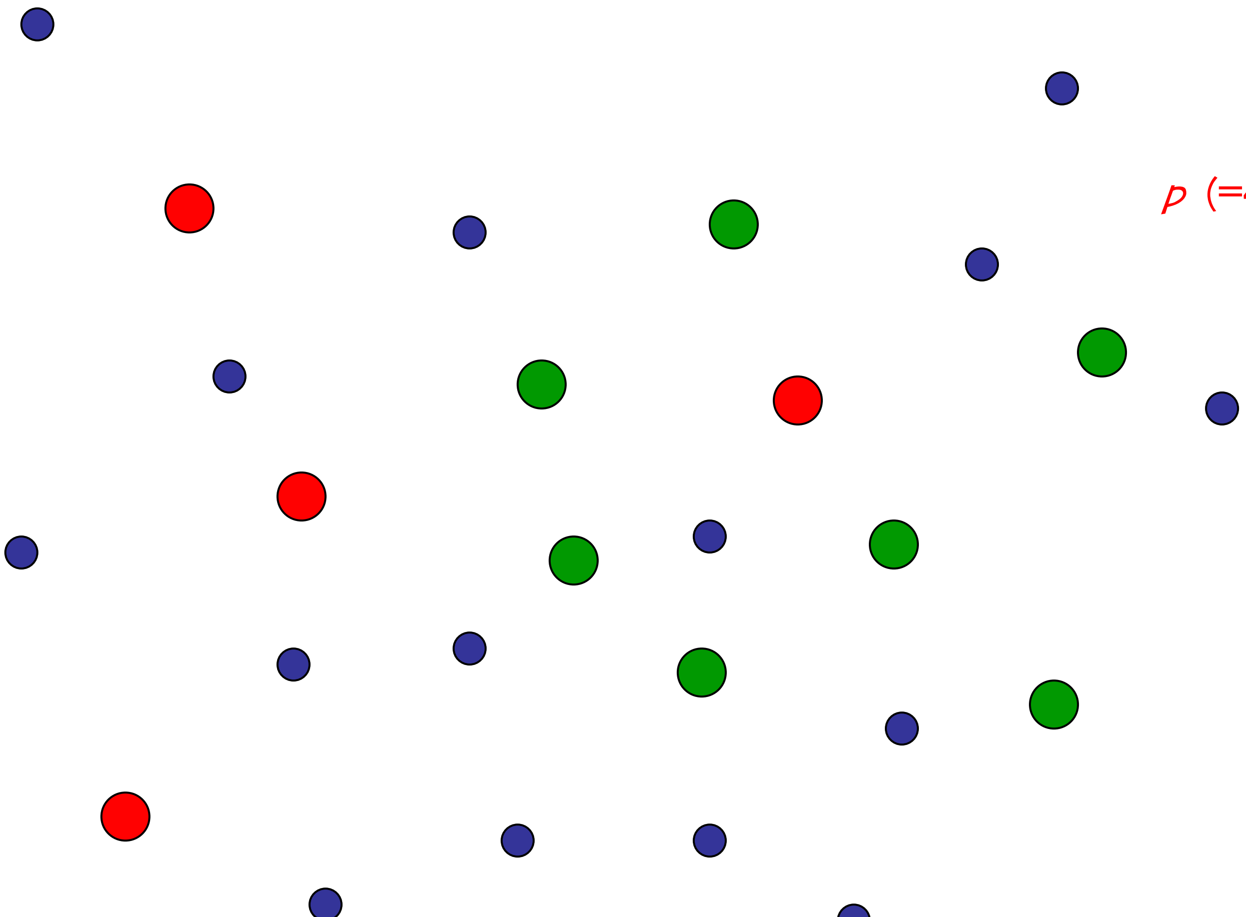
p-median problem



n (=11) potential service locations

m (=15) customers

p-median problem

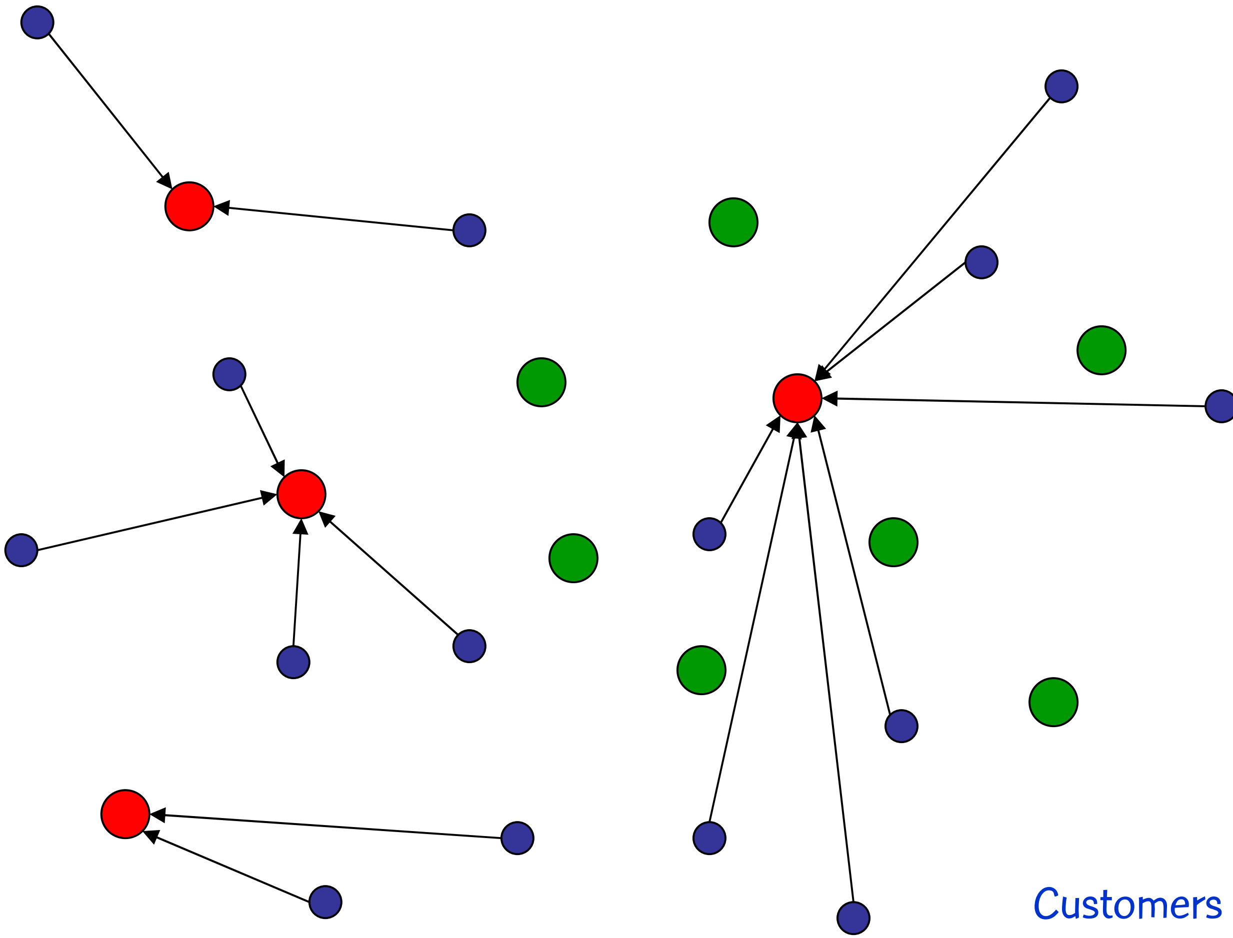


p (=4) service sites to be opened

n (=11) potential service locations

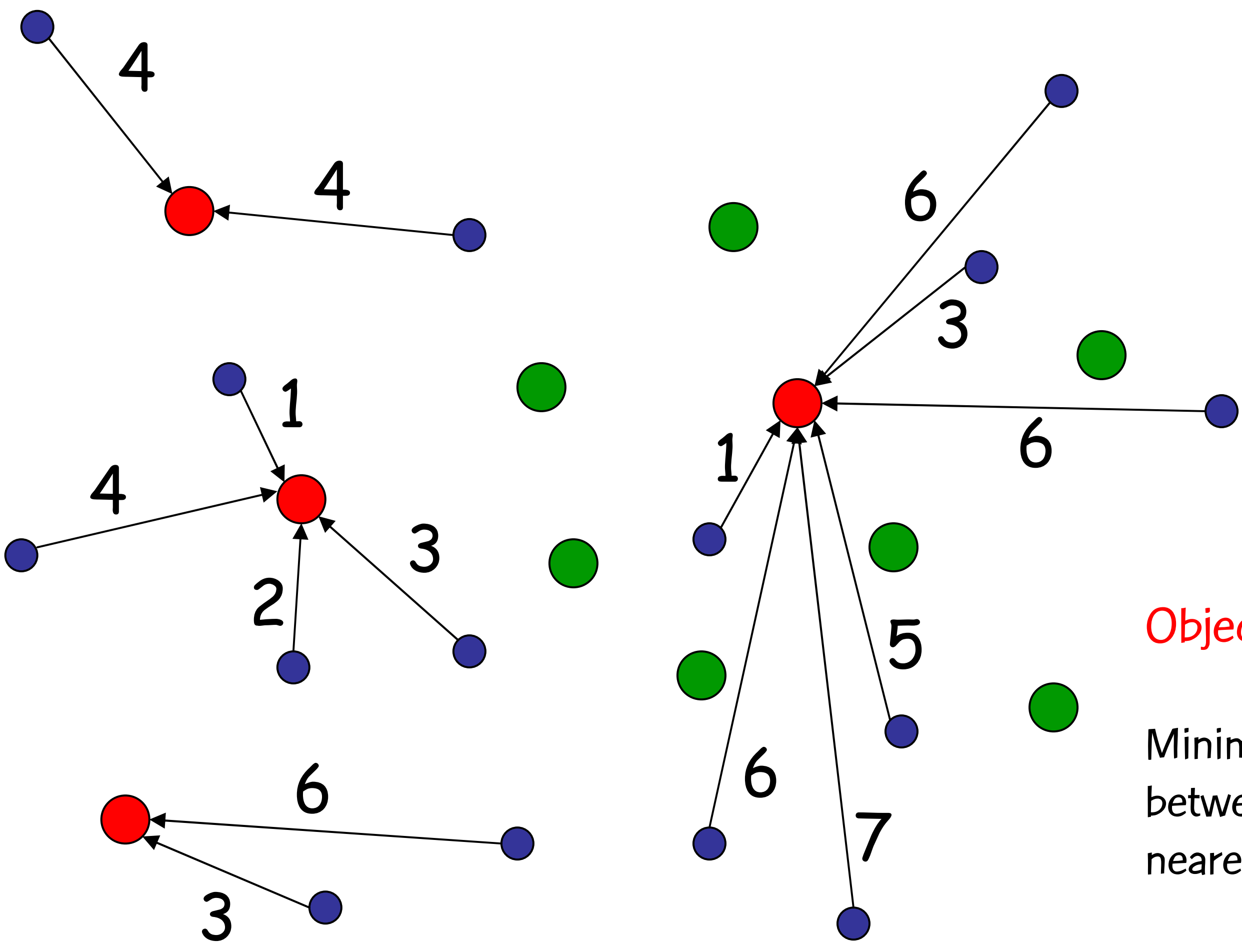
m (=15) customers

p-median problem



Customers home into nearest open service center.

p-median problem

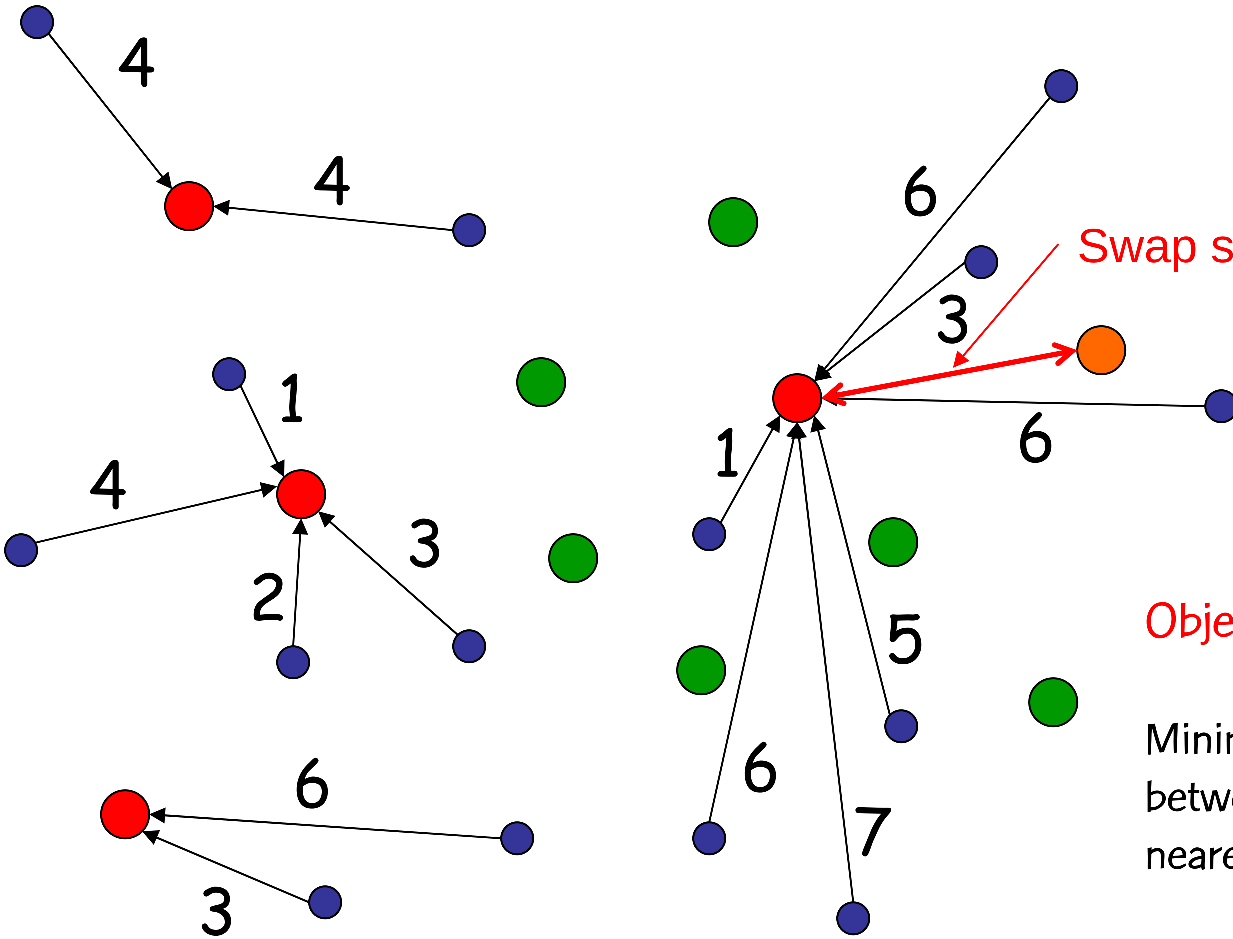


Objective of optimization:

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

Total distance = 61

p-median problem

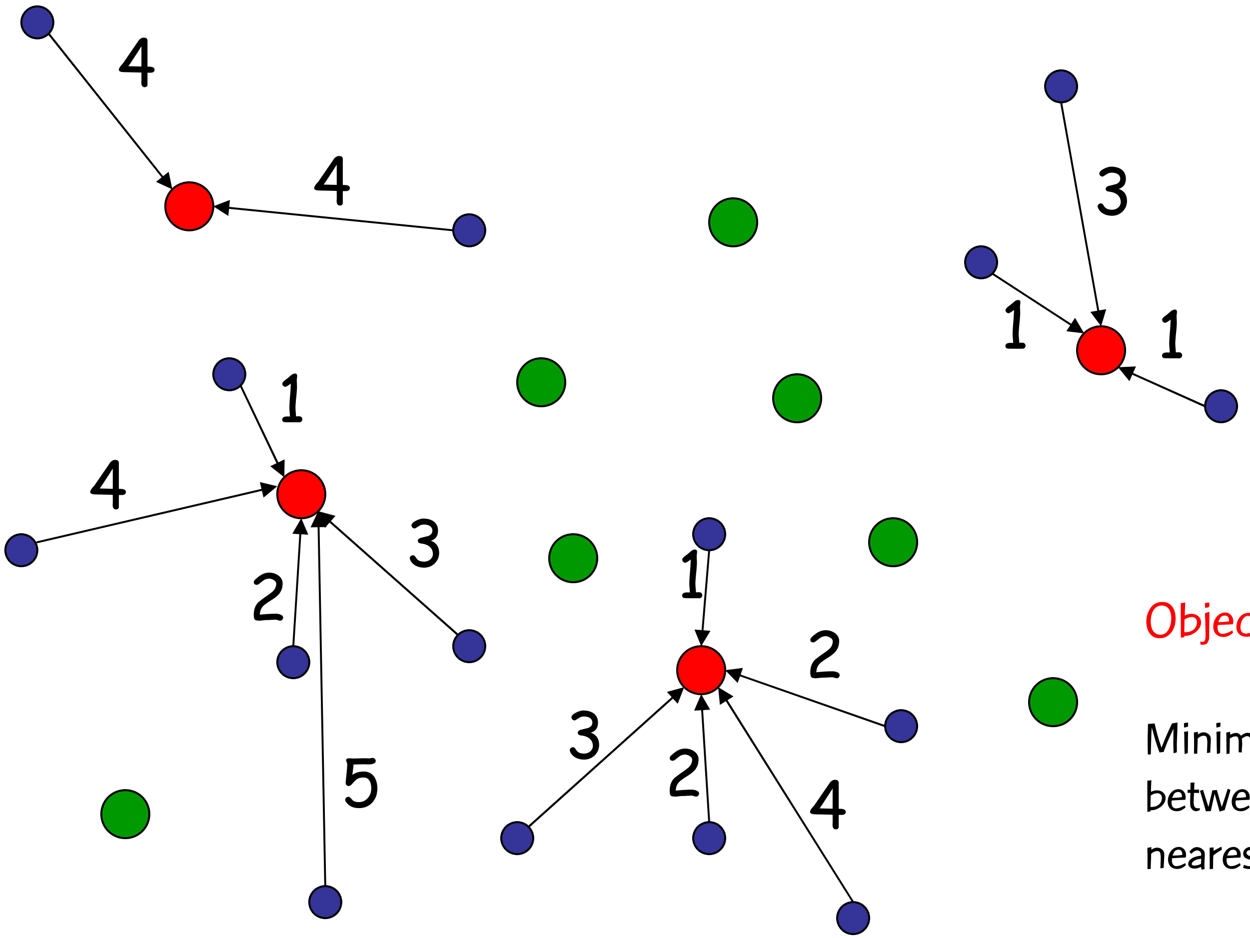


Objective of optimization:

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

Total distance = 61

p-median problem



Objective of optimization:

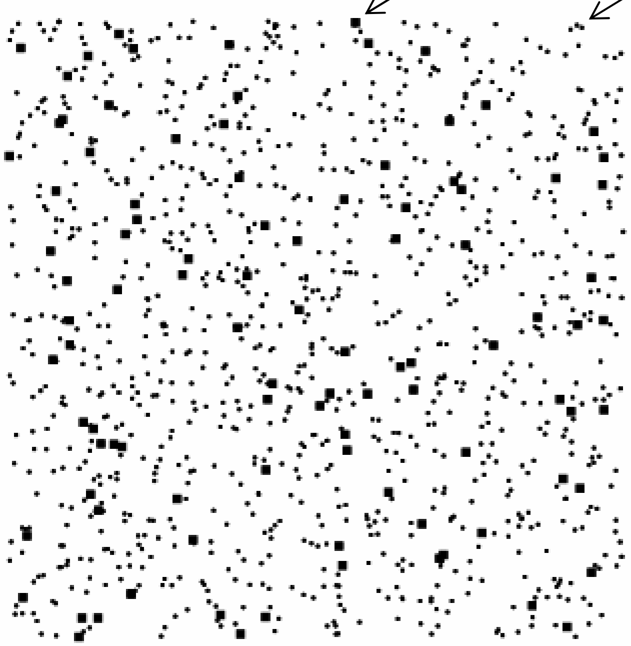
Minimize sum of the distances between customers and their nearest open service center.

Total distance = 40 < 61

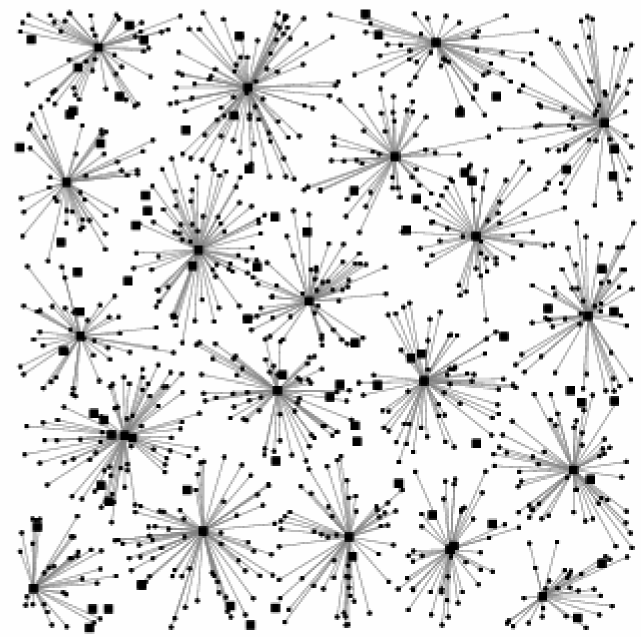
Example: 1000 customer locations, choose best 20 of 100 service locations

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

- Introduced in Teitz and Bart (1968).
- 5-opt for metric cases (Arya et al. , 2001)
- 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

AT&T

Swap based local search

Notation:

$$\phi_1(u) = \text{be the facility closest to } u$$
$$\phi_2(u) = \text{be the facility second closest to } u$$
$$d(u,v) = \text{distance between } u \text{ and } v$$
$$d_1(u) = d(u, \phi_1(u))$$
$$d_2(u) = d(u, \phi_2(u))$$

$$profit(f_i, f_r) = \sum_{u:\phi_1(u) \neq f_r} \max\{0, [d_1(u) - d(u, f_i)]\} - \sum_{u:\phi_1(u) = f_r} [\min\{d_2(u), d(u, f_i)\} - d_1(u)].$$

(f_i: in, f_r: out)

Complexity of local search:
O(pmn) total time

Whitaker's algorithm (1983): Given facility f_i to swap in, finds facility f_r to swap out in $\Theta(n)$ time.

function `findOut` (S, f_i, ϕ_1, ϕ_2)
    $gain \leftarrow 0$; /* gain resulting from the addition of f_i */
    forall $(f \in S)$ **do** $netloss(f) \leftarrow 0$; /* loss resulting from removal of f */
    forall $(u \in U)$ **do**
        if $(d(u, f_i) \leq d_1(u))$ **then** /* gain if f_i is close enough to u */
            $gain \overset{+}{\leftarrow} [d_1(u) - d(u, f_i)]$;
        else /* loss if facility that is closest to u is removed */
            $netloss(\phi_1(u)) \overset{+}{\leftarrow} \min\{d(u, f_i), d_2(u)\} - d_1(u)$;
        endif
    endforall
    $f_r \leftarrow \text{argmin}_{f \in S}\{netloss(f)\}$;
    $profit \leftarrow gain - netloss(f_r)$;
    return $(f_r, profit)$;
end `findOut`

Complexity of swap-based local search is reduced to O(mn)

Whitaker's observation: Profit can be decomposed into two components, which we call gain and netloss.

Whitaker's algorithm

- Whitaker computes gain and netloss to determine profit of a swap:

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

Complexity to compute all profits: $O(mn)$

Our algorithm

- 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

Our algorithm

We have a paper describing the local search algorithm:

M.G.C. Resende and R.F. Werneck, *A fast swap-based local search procedure for location problems*, AT&T Labs Research Technical Report TD-5R3KBH, Florham Park, NJ, Sept. 2003.

http://www.research.att.com/~mgcr/doc/locationls.pdf

AT&T

Our algorithm

- Defines gain like Whitaker:

$$gain(f_i) = \sum_{u \in U} \max\{0, d_1(u) - d(u, f_i)\}$$

Decrease in solution value if f_i is added, assuming no facility is removed.

- But computes netloss indirectly, by using loss:

$$loss(f_r) = \sum_{u:\phi_1(u)=f_r} [d_2(u) - d_1(u)]$$

Increase in solution value if f_r is removed, assuming no facility is added.

AT&T

Our algorithm

- From Whitaker, we have:

$$netloss(f_i, f_r) = \sum_{\substack{u:[\phi_1(u)=f_r]\wedge \\ [d(u,f_i)>d_1(u)]}} [\min\{d(u, f_i), d_2(u)\} - d_1(u)]$$

- For all pairs $\{f_i, f_r\}$, we define:

$$extra(f_i, f_r) = loss(f_r) - netloss(f_i, f_r)$$

Substituting loss and netloss into the expression for extra, (after some algebra) we get ...

AT&T

Our algorithm

- Our final expression for extra:

$$extra(f_i, f_r) = \sum_{\substack{u:[\phi_1(u)=f_r]\wedge \\ [d(u,f_i)<d_2(u)]}} [d_2(u) - \max\{d(u, f_i), d_1(u)\}] \geq 0$$

- And now, we can compute the profit of swapping f_i in and f_r out:

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

Our algorithm

- So we have to compute three structures:

$$loss(f_r) = \sum_{u:\phi_1(u)=f_r} [d_2(u) - d_1(u)]$$

$$gain(f_i) = \sum_{u \in U} \max\{0, d_1(u) - d(u, f_i)\}$$

$$extra(f_i, f_r) = \sum_{\substack{u:[\phi_1(u)=f_r]\wedge \\ [d(u,f_i)<d_2(u)]}} [d_2(u) - \max\{d(u, f_i), d_1(u)\}]$$

- Each of them is a summation over the set of users:

The contribution of each user can be computed independently.

AT&T

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:
  - 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
  - more complicated
  - uses 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

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
```

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

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
```

AT&T

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).

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 (recall extra is always nonnegative)
  - Compute the exact profit only for pairs that have extra greater than zero.

AT&T

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.

AT&T

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.

AT&T

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.

Local search 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.

Local search 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.

Local search 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.

Local search 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.

Local search 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.

Local search 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, ...).

Local search 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*.

AT&T

Local search results

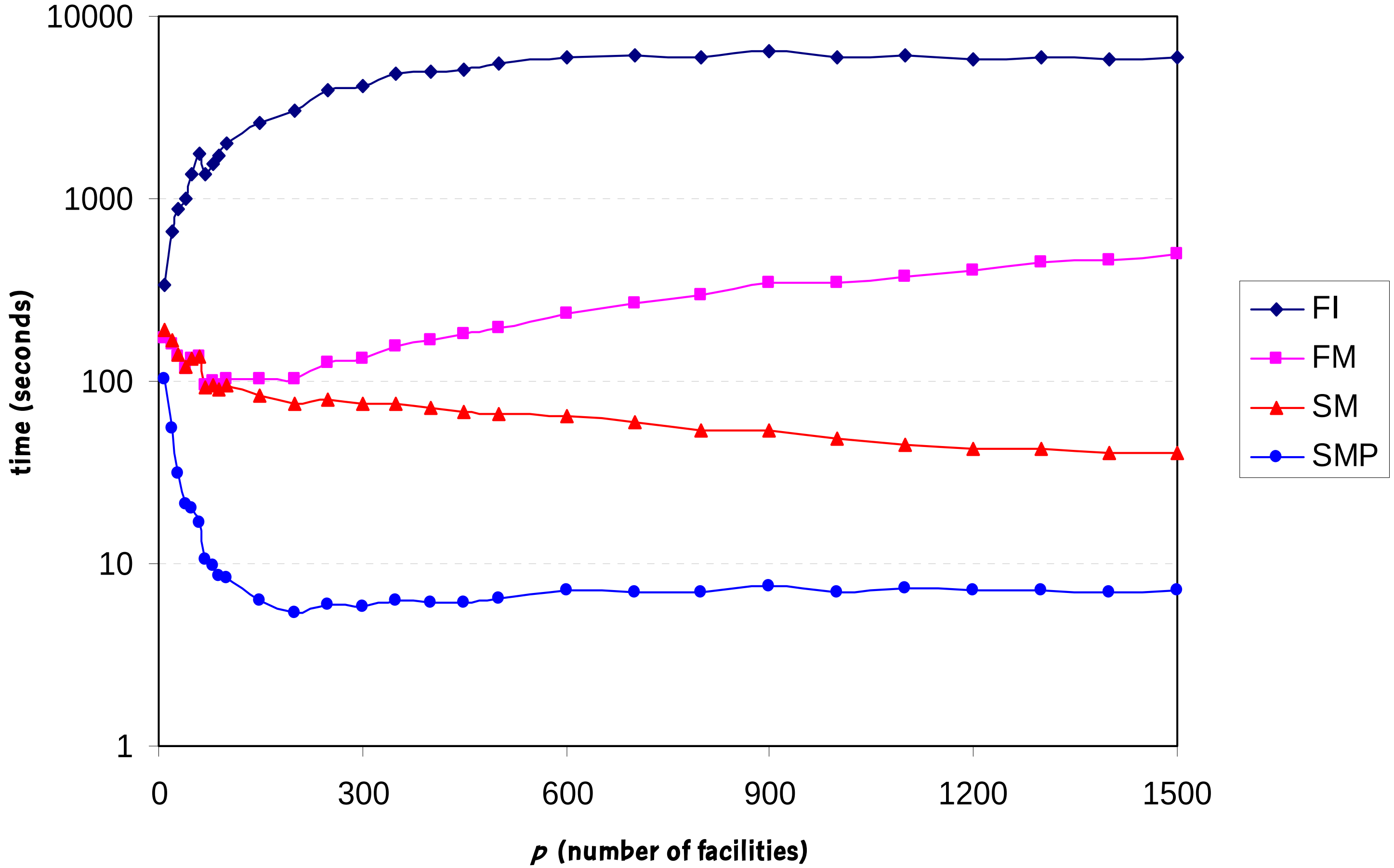
Speedups w.r.t. Whitaker's **FI** (worst cases):

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.

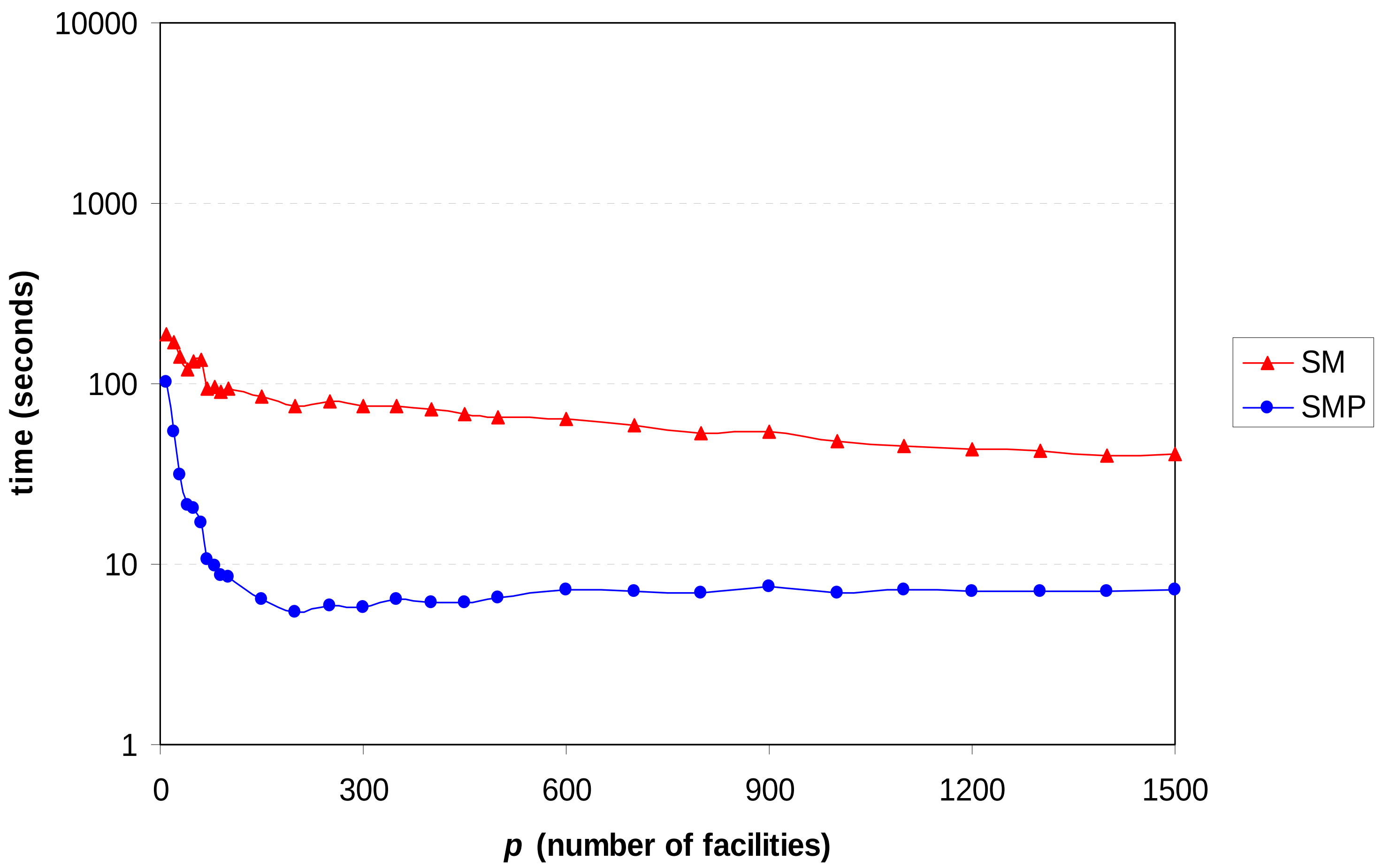
Local search results



Largest instance tested: 5934 users, Euclidean.

(preprocessing times not considered)

Local search results



Note that preprocessing significantly accelerates the algorithm.

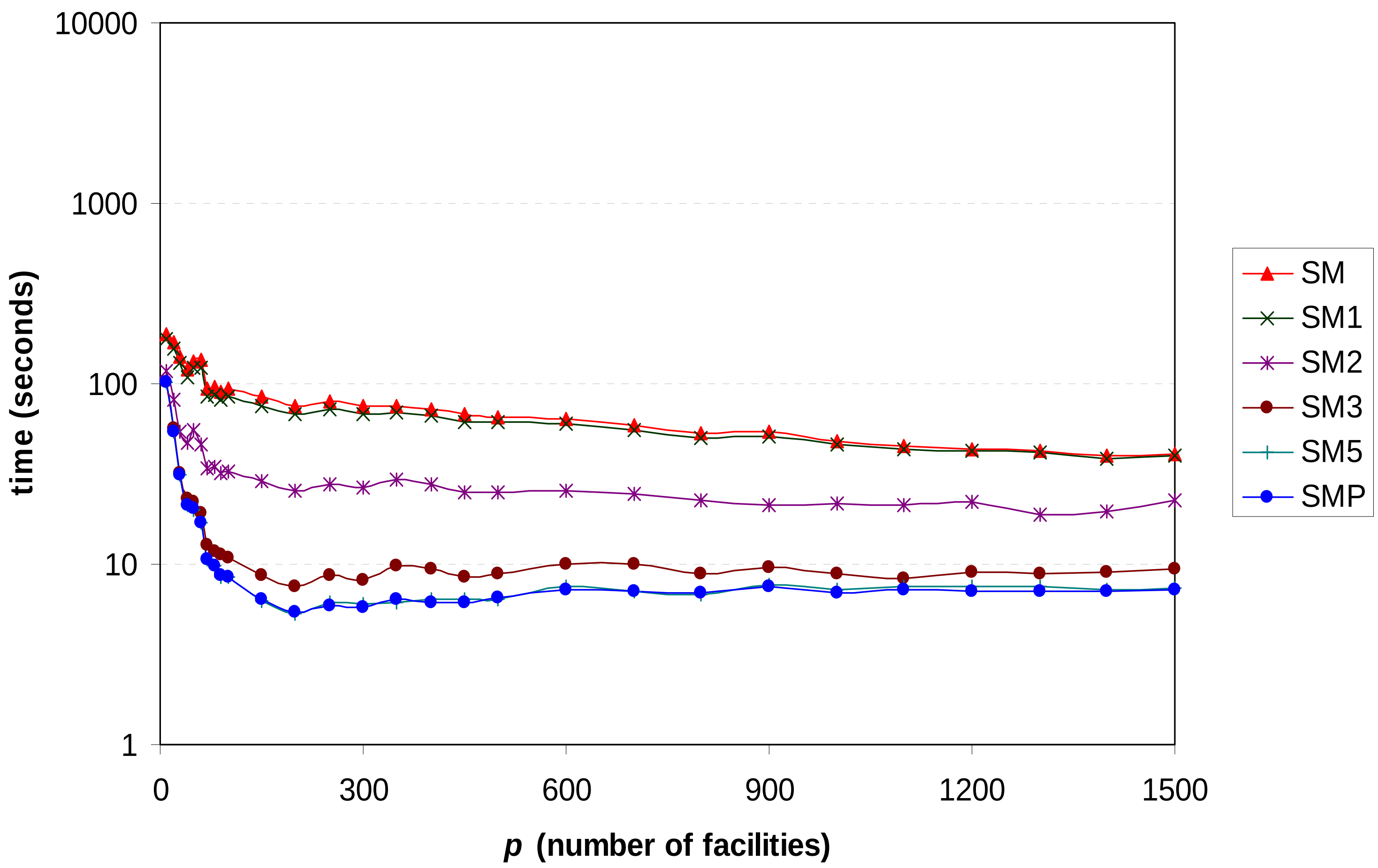
AT&T

Local search 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.

AT&T

Local search 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 next show results of a GRASP with path-relinking based on this local search.
  - Other existing methods may benefit from it.

GRASP: greedy randomized adaptive search procedure

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

Paper

We have a paper describing the GRASP with path-relinking (hybrid algorithm):

M.G.C. Resende and R.F. Werneck, *A hybrid heuristic for the p-median problem*, AT&T Labs Research Technical Report TD-5NWRCR, Florham Park, NJ, June 2003.

http://www.research.att.com/~mgcr/doc/hhpmedian.pdf

AT&T

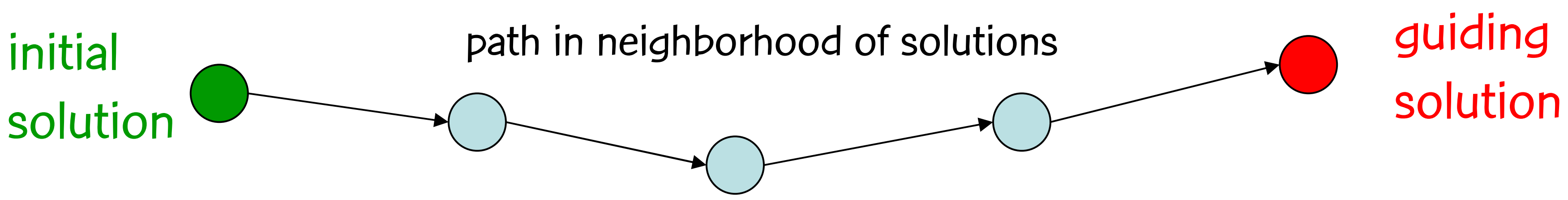
Sample greedy construction

- 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 $q = \lceil \log_2 (m / p) \rceil$.

AT&T

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

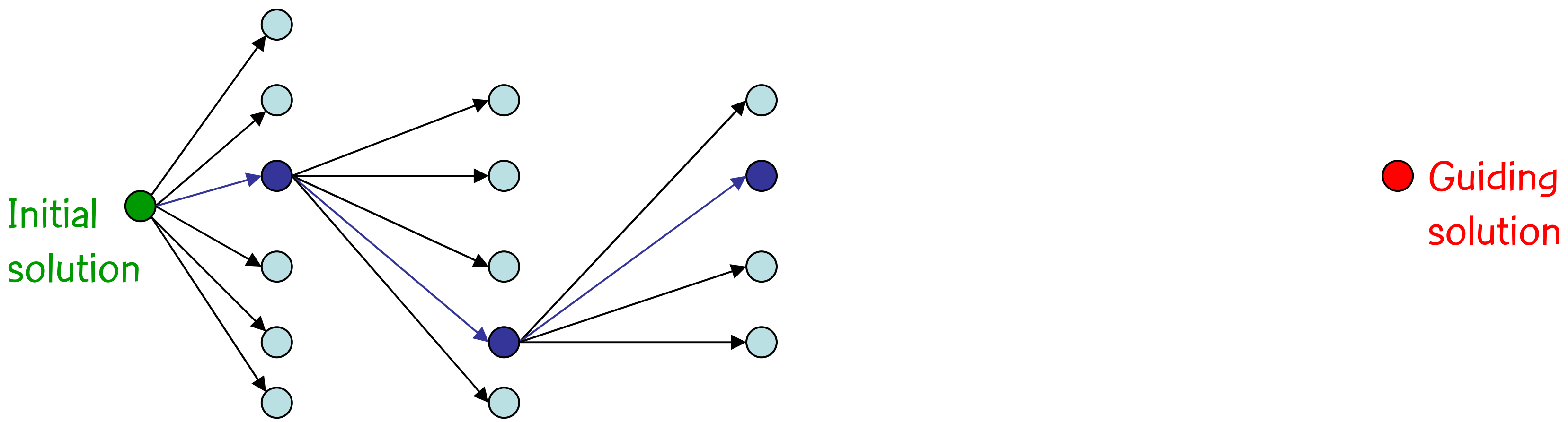
- Path is generated by selecting moves that introduce in the initial solution attributes of the guiding solution.
- At each step, all moves that incorporate attributes of the guiding solution are analyzed and best move is taken.

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.

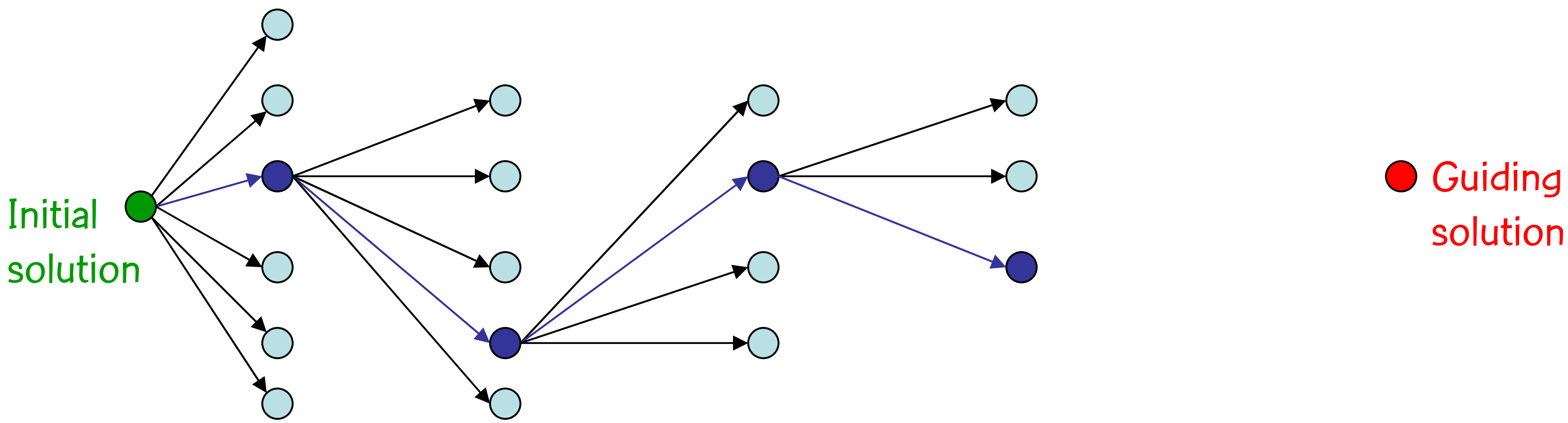
Path-relinking

- 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.



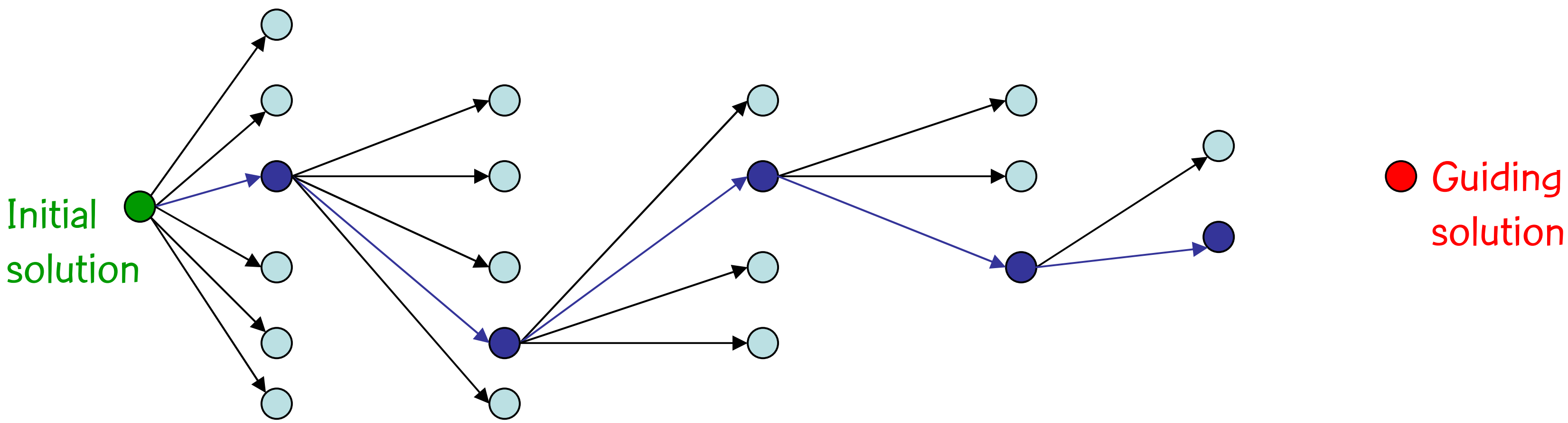
Path-relinking

- 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.



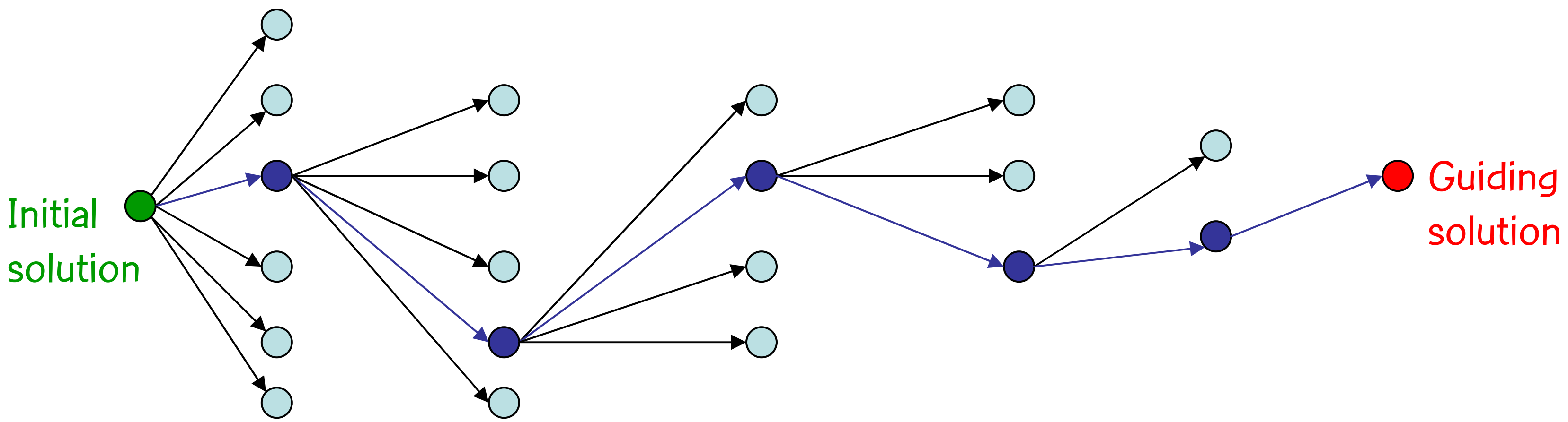
Path-relinking

- 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 & 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).

AT&T

GRASP & path-relinking

- Use the solution GRASP iterate, produced after construction and local search, as the initial solution.
- Use a solution selected at random from the set of elite solutions as the target solution.

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

AT&T

Results: Test problems

- TSP: Set of points on the plane (74 instances with 1400, 3038, and 5934 nodes)
  - 1400 node instance: p = 10, 20, ... 450, 500
  - 3038 node instance: p = 10, 20, ... 950, 1000
  - 5934 node instance: p = 10, 20, ... 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.

AT&T

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

AT&T

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

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.

AT&T

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.

Local search was also applied to uncapacitated facility location problem

- Consistently outperforms other heuristics in the literature.
- Paper: M.G.C. Resende and R.F. Werneck, *A hybrid multi-start heuristic for the uncapacitated facility location problem*, AT&T Labs Research Technical Report TD-5RELRR, Florham Park, NJ, Sept. 2003.

http://www.research.att.com/~mgcr/doc/guflp.pdf

AT&T

Software availability

Our software (local search, and hybrid heuristics for p-median and facility location) as well as all test instances used in our studies are available for download at:

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