Short course on Greedy Randomized Adaptive Search Procedures:

GRASP

XXXIX Simpósio Brasileiro de Pesquisa Operacional
Fortaleza, Brazil ~ August 29-31, 2007

Mauricio G. C. Resende
AT&T Labs Research
Florham Park, New Jersey
mgcr@att.com

Summary

- Day 1
  - Combinatorial opt. & metaheuristics
  - Local search
  - Greedy algorithm
  - Basic GRASP
    - Construction
    - Local search within GRASP
  - Some extensions
    - Reactive GRASP
    - Memory in construction

- Day 2
  - Prob. distribution of running time
  - Time-to-target plots
  - Path-relinking (PR) & Evolutionary PR (EvPR)
  - GRASP with PR
  - GRASP with EvPR
  - Parallel GRASP
    - Independent threads
    - Cooperative threads
  - Implementation & testing

Day 1 of Short Course on GRASP

Combinatorial optimization and metaheuristics

Combinatorial Optimization

Handbook of Applied Optimization

P.M. Pardalos and M.G.C. Resende, eds. Oxford U. Press, 2002

Combinatorial optimization: process of finding the best, or optimal, solution for problems with a discrete set of feasible solutions.

Applications: e.g. routing, scheduling, packing, inventory and production management, location, logic, and assignment of resources.

Economic impact: e.g. transportation (airlines, trucking, rail, and shipping), forestry, manufacturing, logistics, aerospace, energy (electrical power, petroleum, and natural gas), agriculture, biotechnology, financial services, and telecommunications.

Combinatorial Optimization

- Given:
  - discrete set of solutions X
  - objective function $f(x): x \in X \rightarrow R$
- Objective:
  - find $x \in X : f(x) \leq f(y), \forall y \in X$

Combinatorial Optimization

- Much progress in recent years on finding exact (provably optimal) solution: dynamic programming, cutting planes, branch and cut, ...
- Many hard combinatorial optimization problems are still not solved exactly and require good heuristic methods.
- Aim of heuristic methods for combinatorial optimization is to quickly produce good-quality solutions, without necessarily providing any guarantee of solution quality.

Metaheuristics

- Metaheuristics are high level procedures that coordinate simple heuristics, such as local search, to find solutions that are of better quality than those found by the simple heuristics alone.
- Examples: simulated annealing, genetic algorithms, tabu search, scatter search, ant colony optimization, variable neighborhood search, and GRASP.

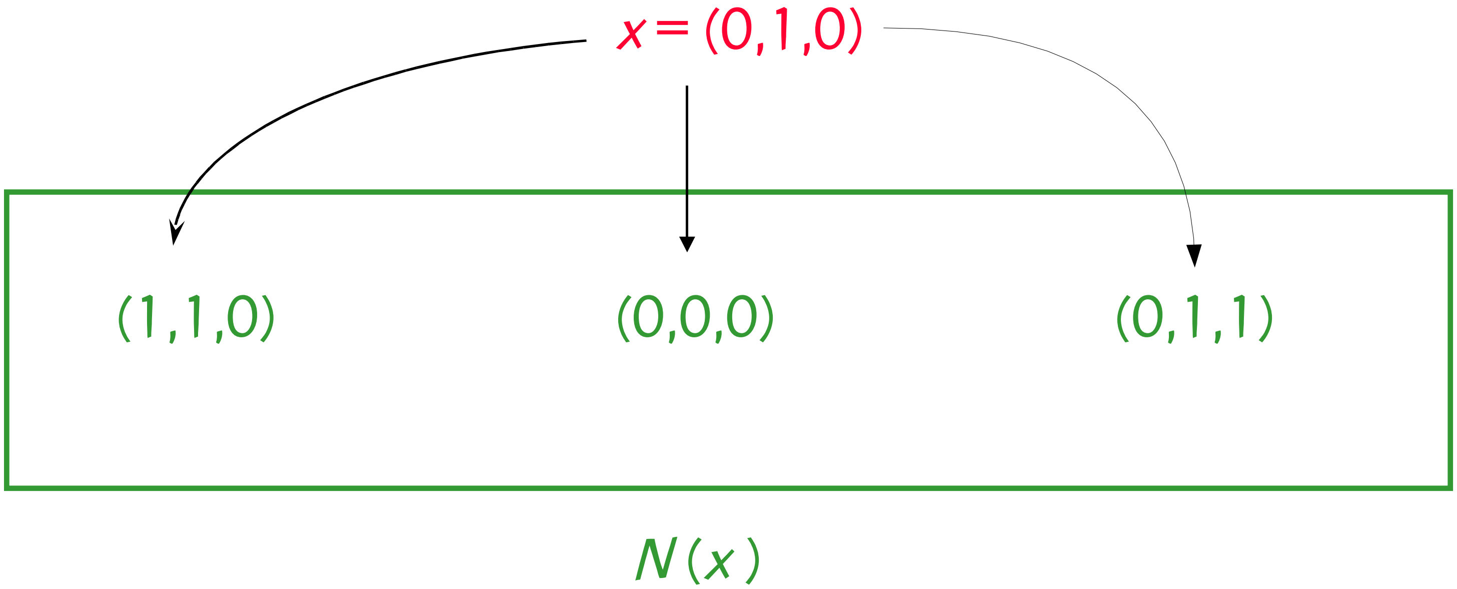
Local search

Local Search

- To define local search, one needs to specify a local neighborhood structure.
- Given a solution x , the elements of the neighborhood N(x) of x are those solutions y that can be obtained by applying an elementary modification (often called a move) to x.

Local Search Neighborhoods

Consider $x = (0,1,0)$ and the 1-flip neighborhood of a 0/1 array.



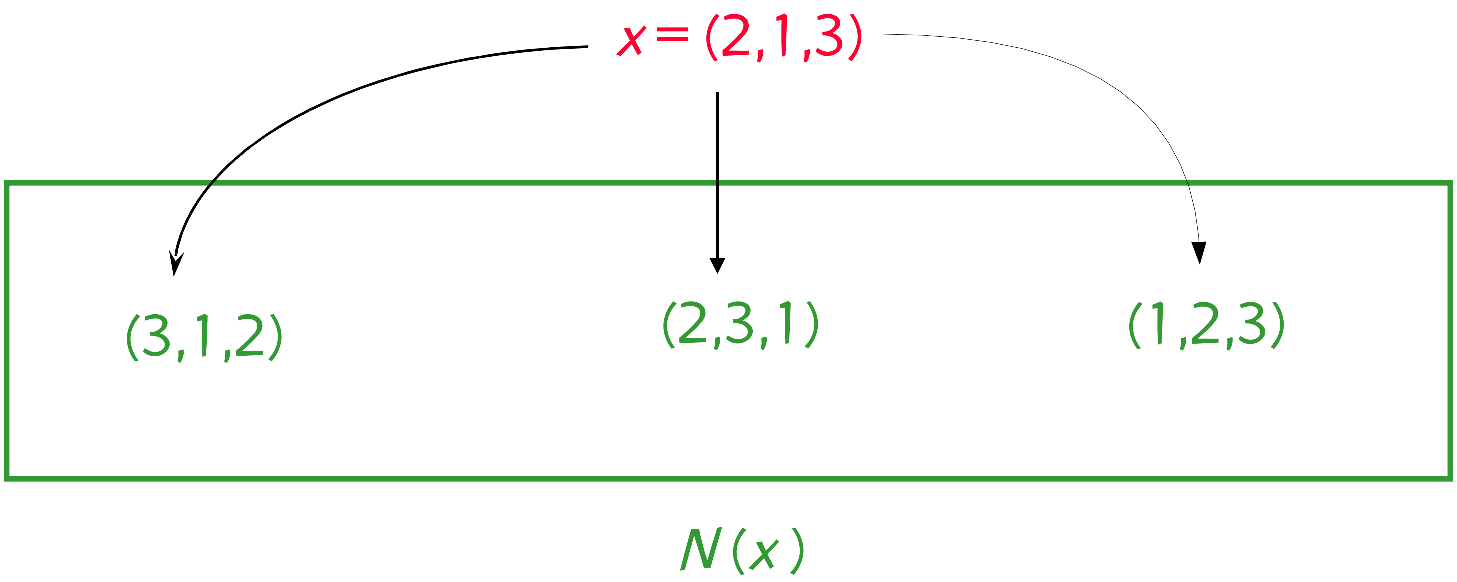
$x = (0,1,0)$

$(1,1,0)$ $(0,0,0)$ $(0,1,1)$

$N(x)$

Local Search Neighborhoods

Consider $x = (2,1,3)$ and the 2-swap neighborhood of a permutation array.



$x = (2,1,3)$

(3,1,2)

(2,3,1)

(1,2,3)

$N(x)$

Local Search

Given an initial solution x_0, a neighborhood $N(x)$, and function $f(x)$ to be minimized:

```
$x = x_0$ ;
while ( $\exists\, y \in N(x) \mid f(y) < f(x)$ ) {
    $x = y$ ;
}
```

check for better solution in neighborhood of *x*

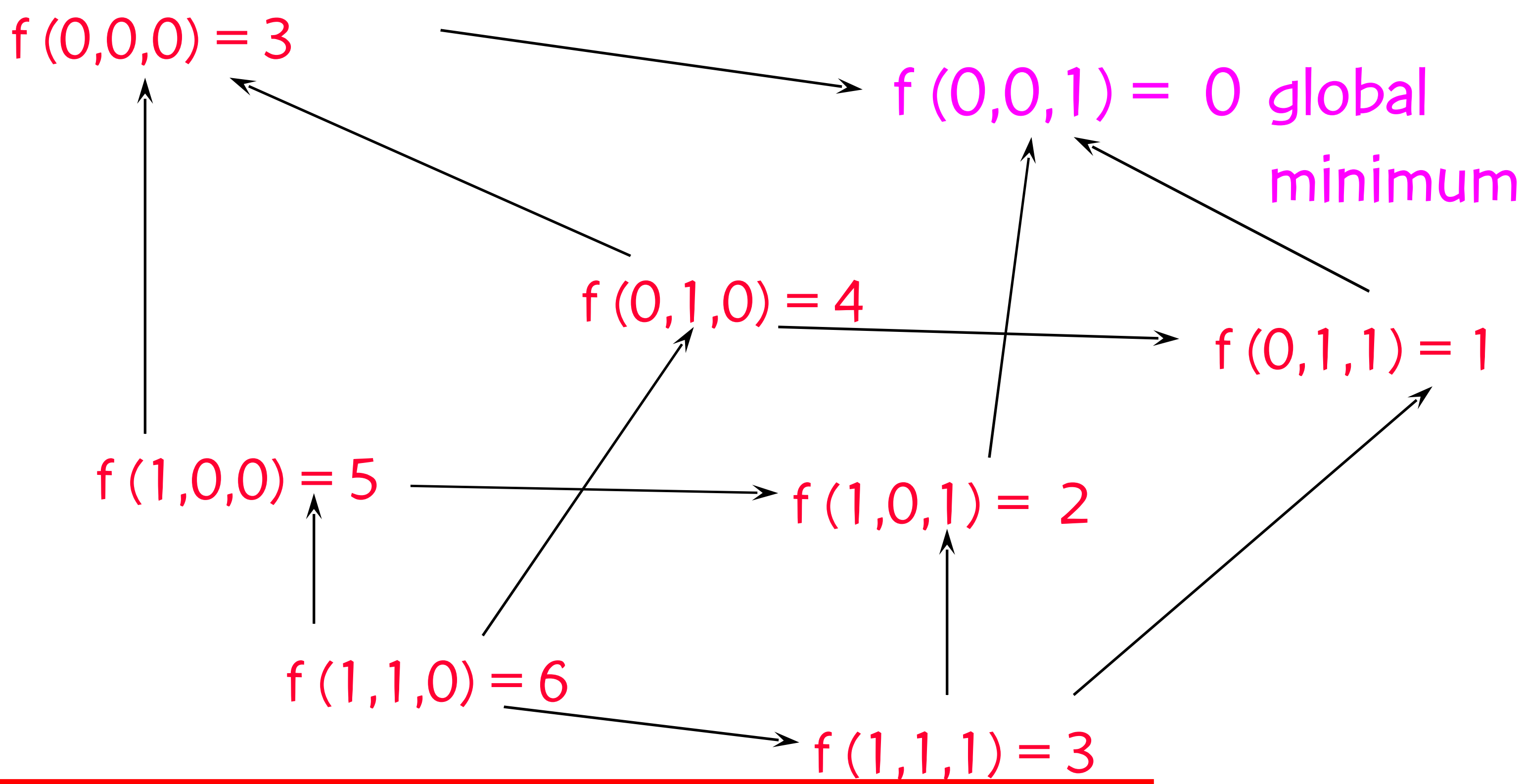
move to better solution *y*

Time complexity of local search can be exponential.

At the end, x is a local minimum of $f(x)$.

Local Search

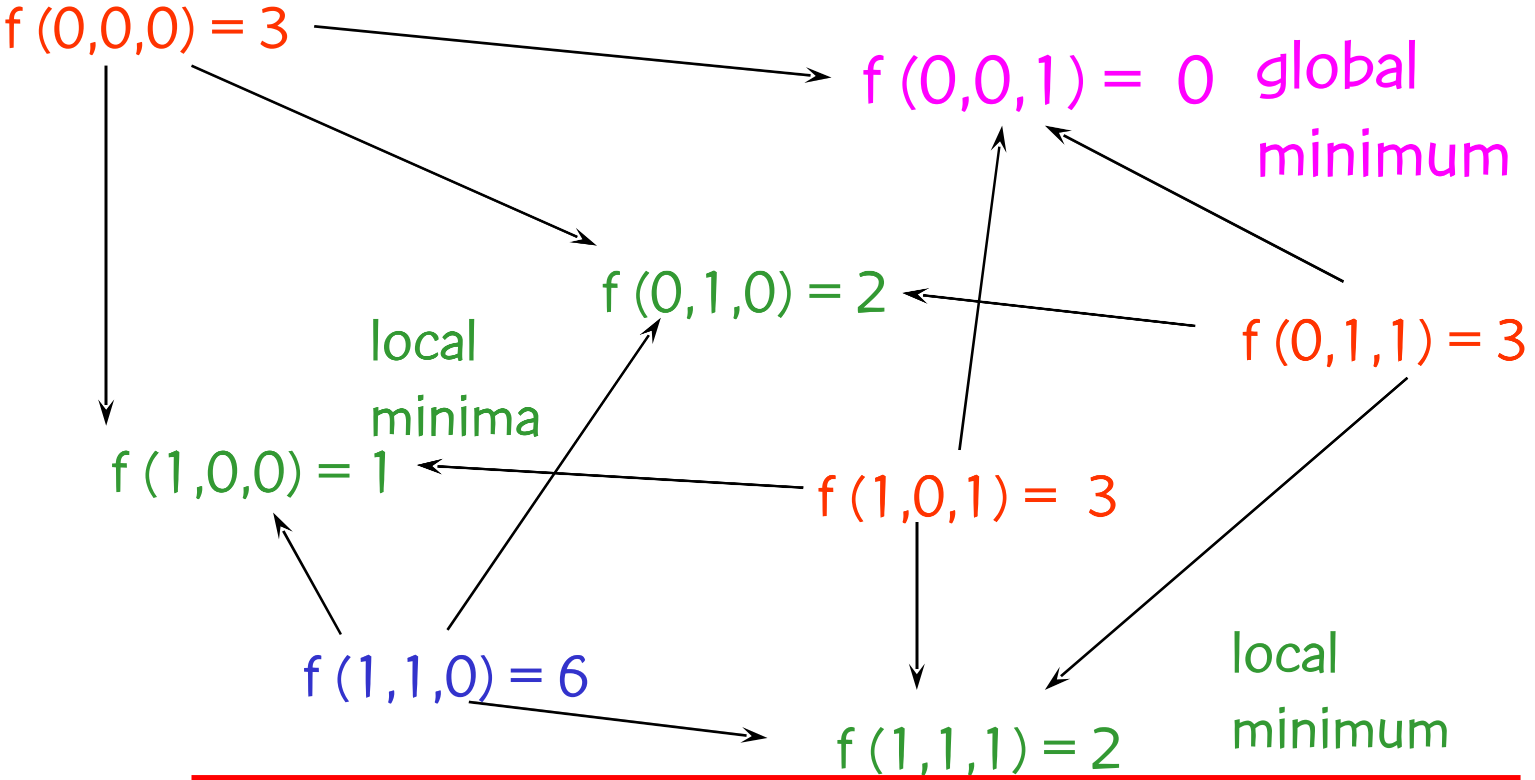
(ideal situation)



With any starting solution Local Search finds the global optimum.

Local Search

(more realistic situation)



f (0,0,0) = 3

f (0,0,1) = 0 global minimum

f (0,1,0) = 2

local minima

f (0,1,1) = 3

f (1,0,0) = 1

f (1,0,1) = 3

f (1,1,0) = 6

f (1,1,1) = 2

local minimum

But some starting solutions lead Local Search to a local minimum.

Local Search

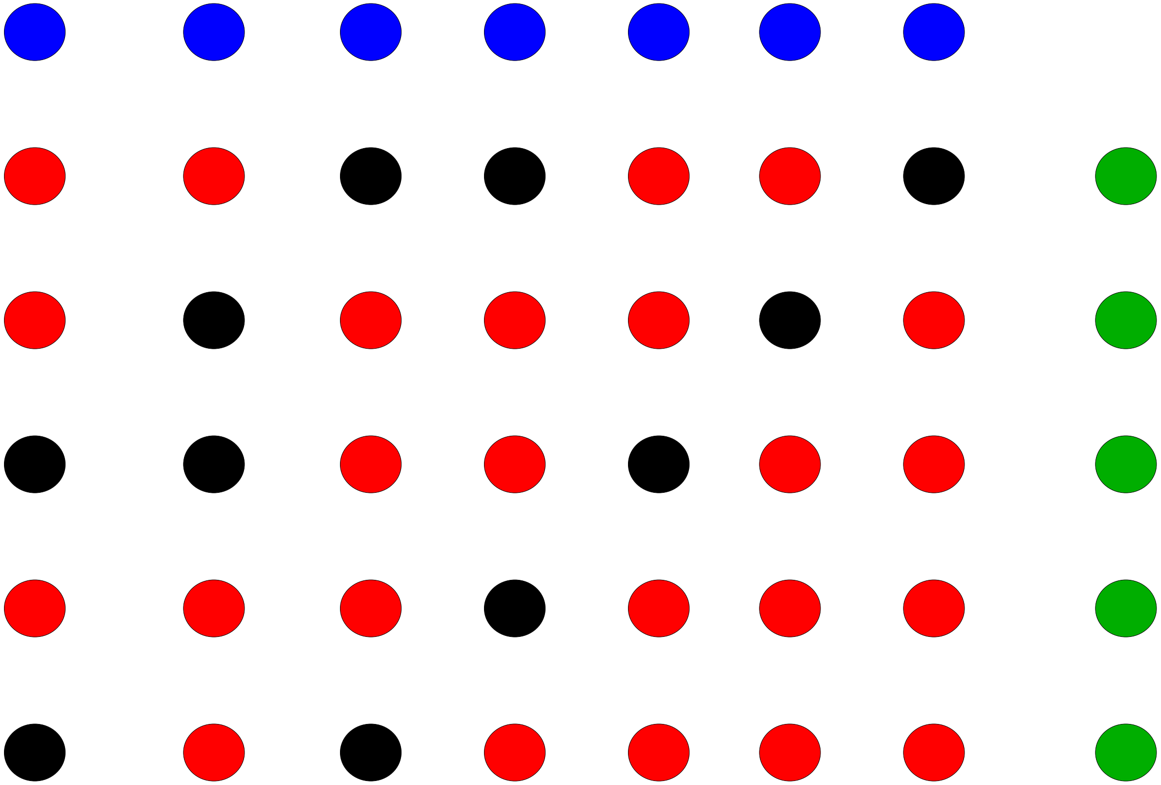
Effectiveness of local search depends on several factors:

- neighborhood structure
- function to be minimized
- starting solution

usually pre-determined

usually easier to control

Example of local search: set covering

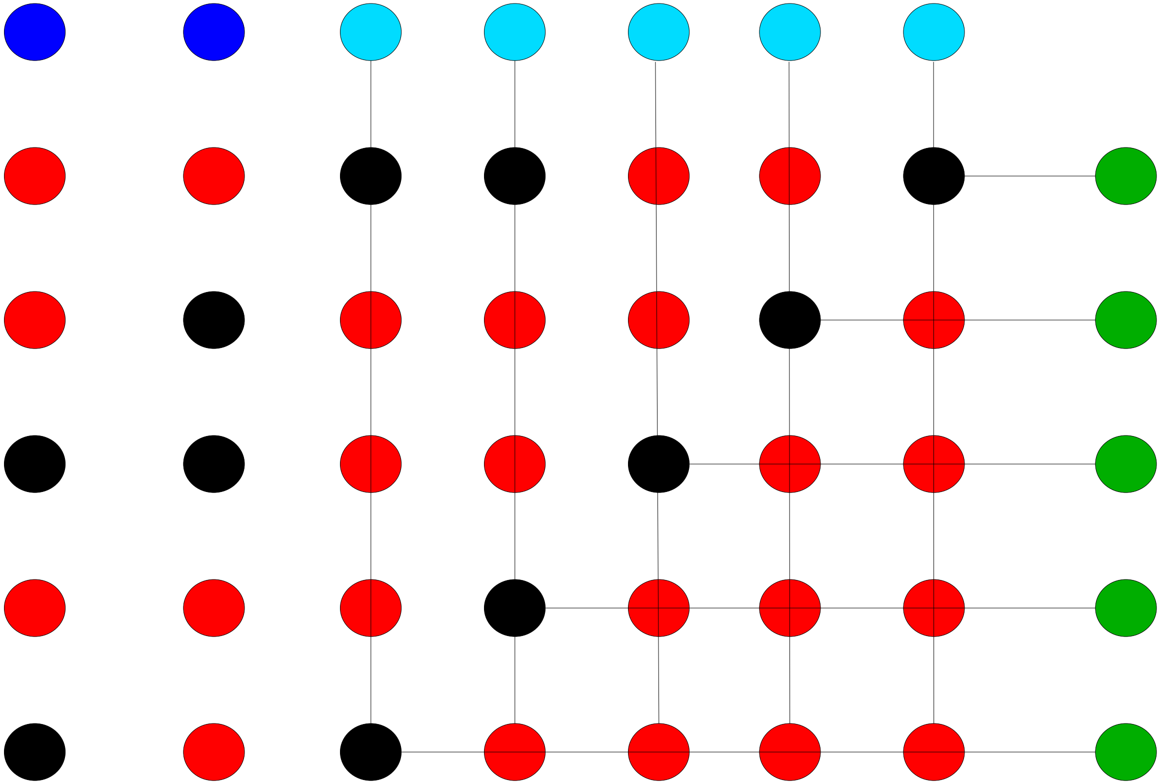


Black indicates column covers row.

Objective: Select smallest subset of columns such that all rows are covered.

Example of local search: set covering

Covering of size 5

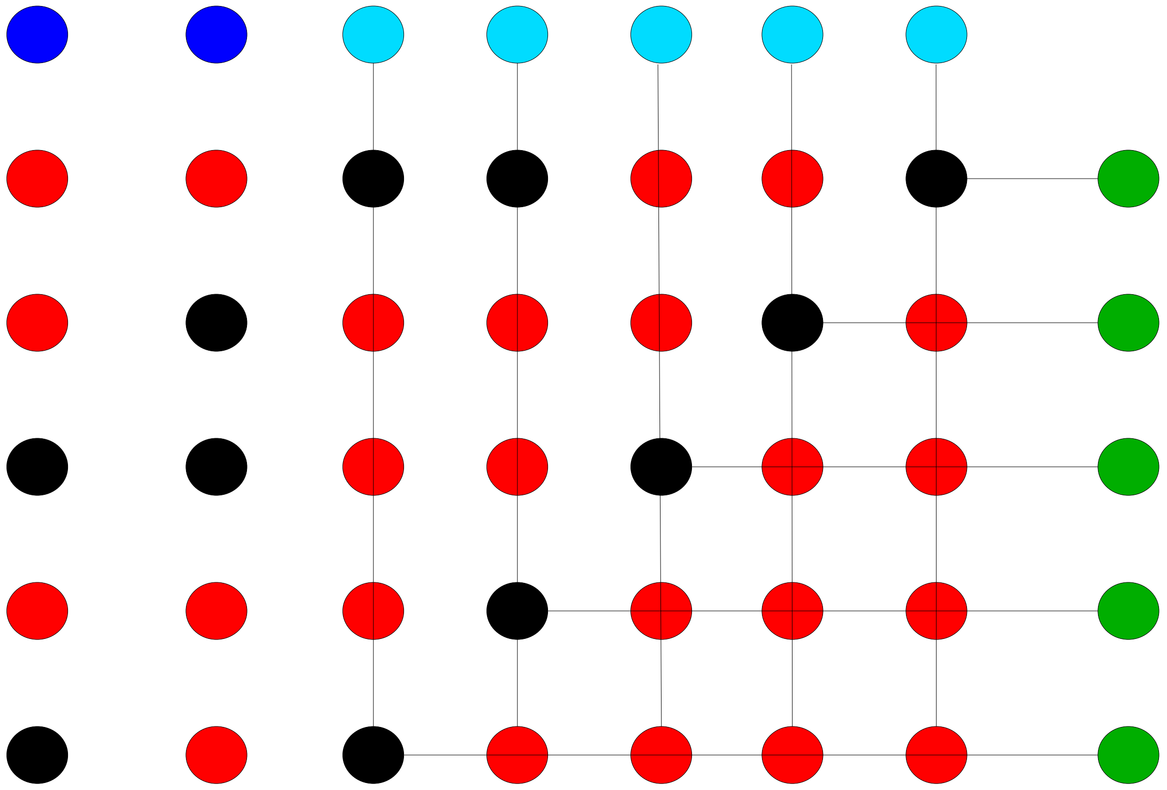


Black indicates column covers row.

Objective: Select smallest subset of columns such that all rows are covered.

Example of local search: set covering

Covering of size 5

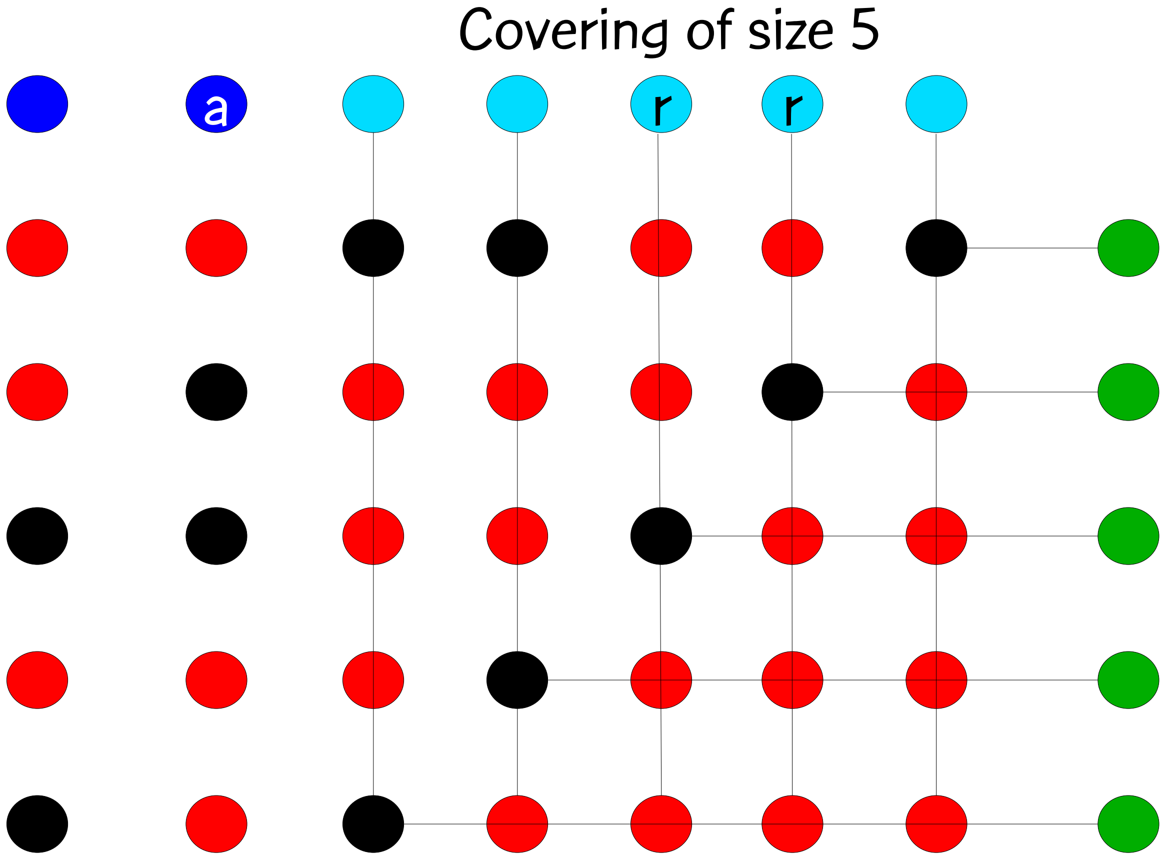


Black indicates column covers row.

Objective: Select smallest subset of columns such that all rows are covered.

Neighborhood: (2,1) exchange

Example of local search: set covering



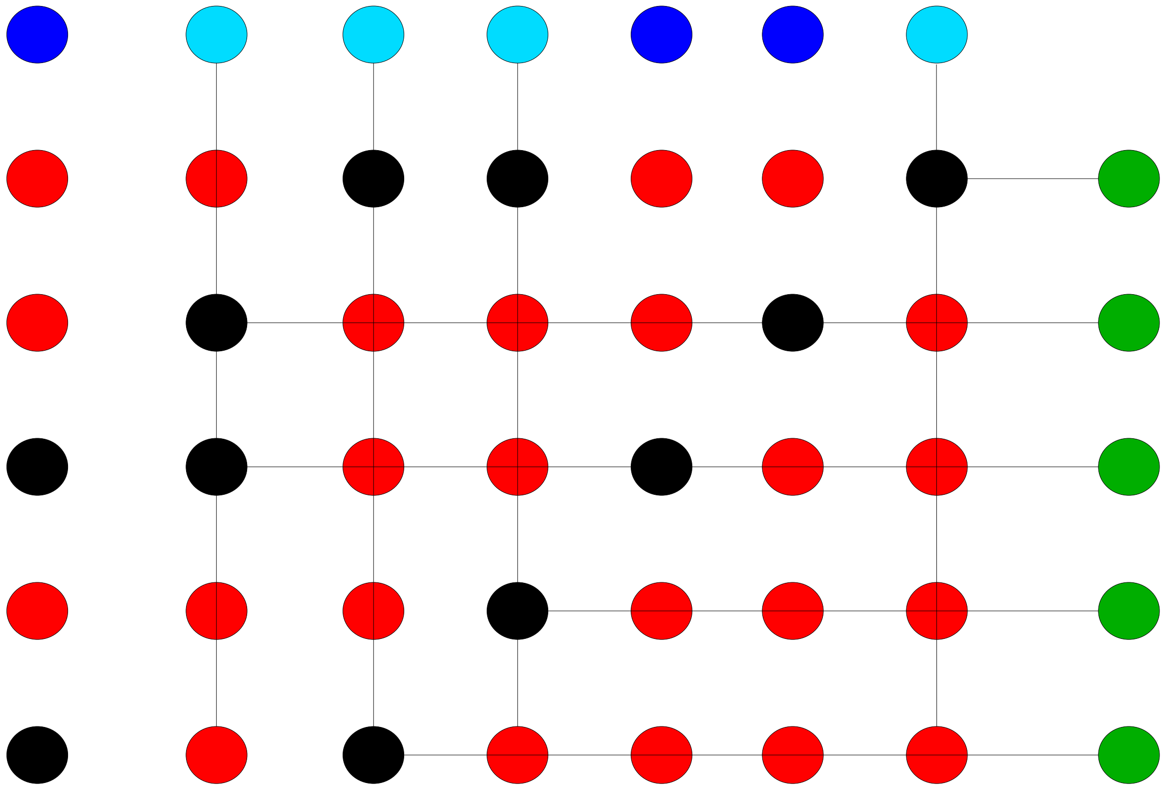
Black indicates column covers row.

Objective: Select smallest subset of columns such that all rows are covered.

Neighborhood: (2,1) exchange

Example of local search: set covering

Covering of size 4



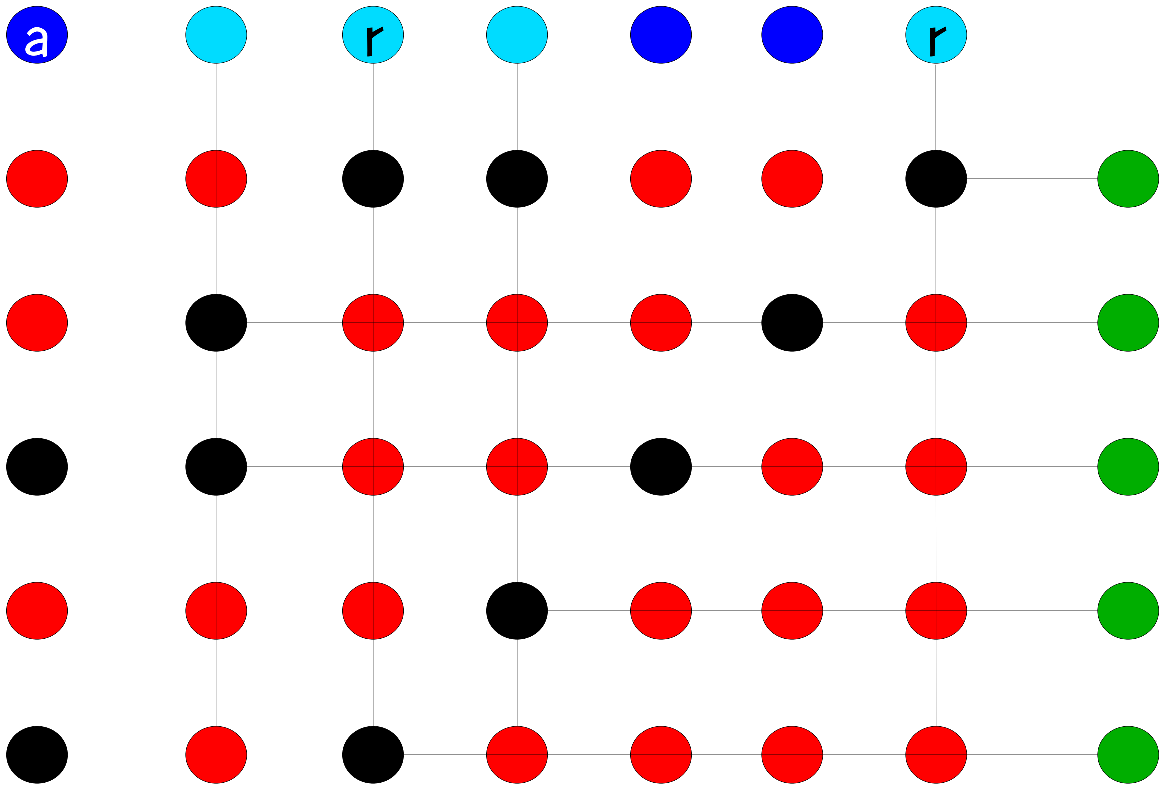
Black indicates column covers row.

Objective: Select smallest subset of columns such that all rows are covered.

Neighborhood: (2,1) exchange

Example of local search: set covering

Covering of size 4

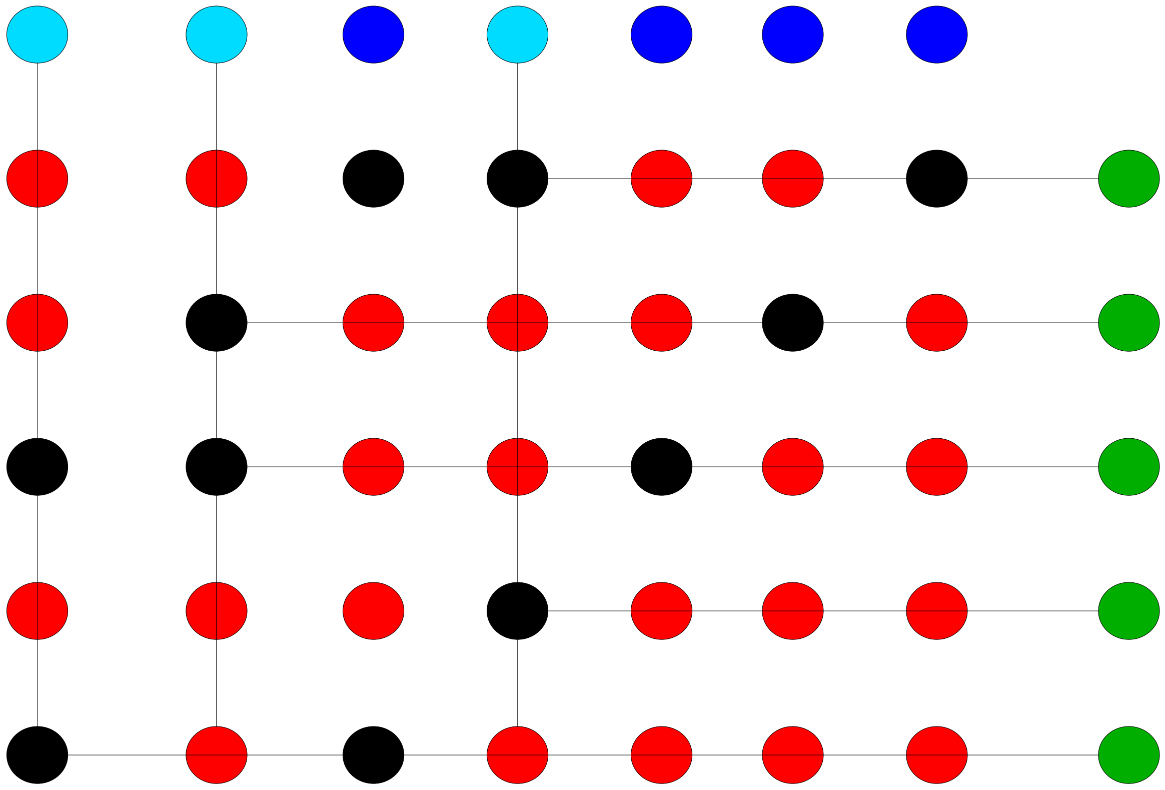


Black indicates column covers row.

Objective: Select smallest subset of columns such that all rows are covered.

Neighborhood: (2,1) exchange

Example of local search: set covering



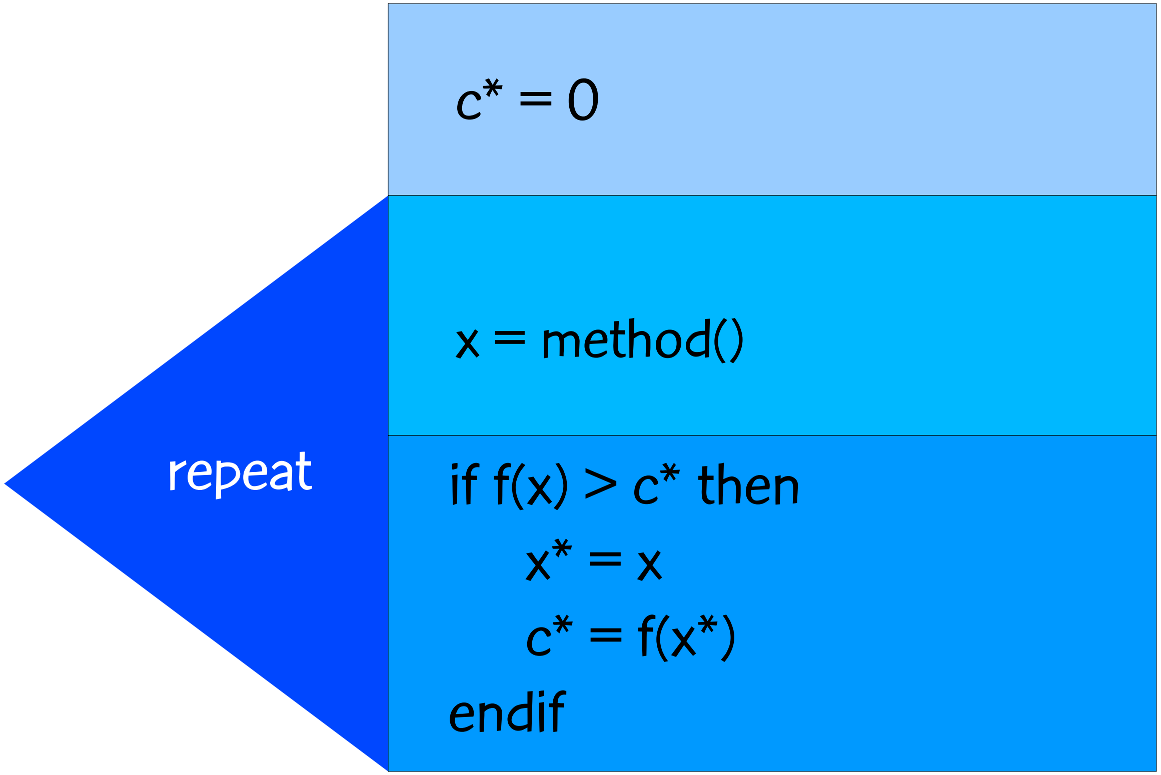
Covering of size 3 (optimal)

Black indicates column covers row.

Objective: Select smallest subset of columns such that all rows are covered.

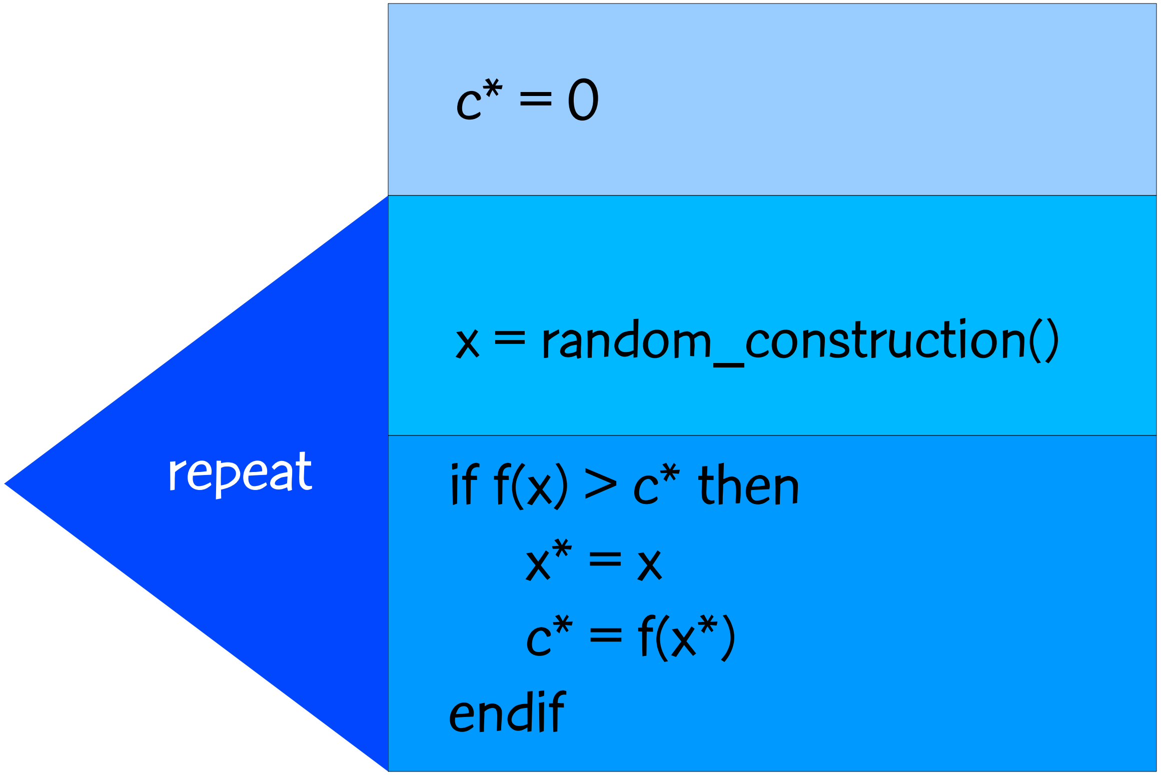
Neighborhood: (2,1) exchange

Multi-start method
(maximization problem)



```
$c^* = 0$
repeat
    $x$ = method()
    if $f(x) > c^*$ then
        $x^* = x$
        $c^* = f(x^*)$
    endif
```

Random multi-start
(maximization problem)



```
c* = 0
repeat
    x = random_construction()
    if f(x) > c* then
        x* = x
        c* = f(x*)
    endif
```

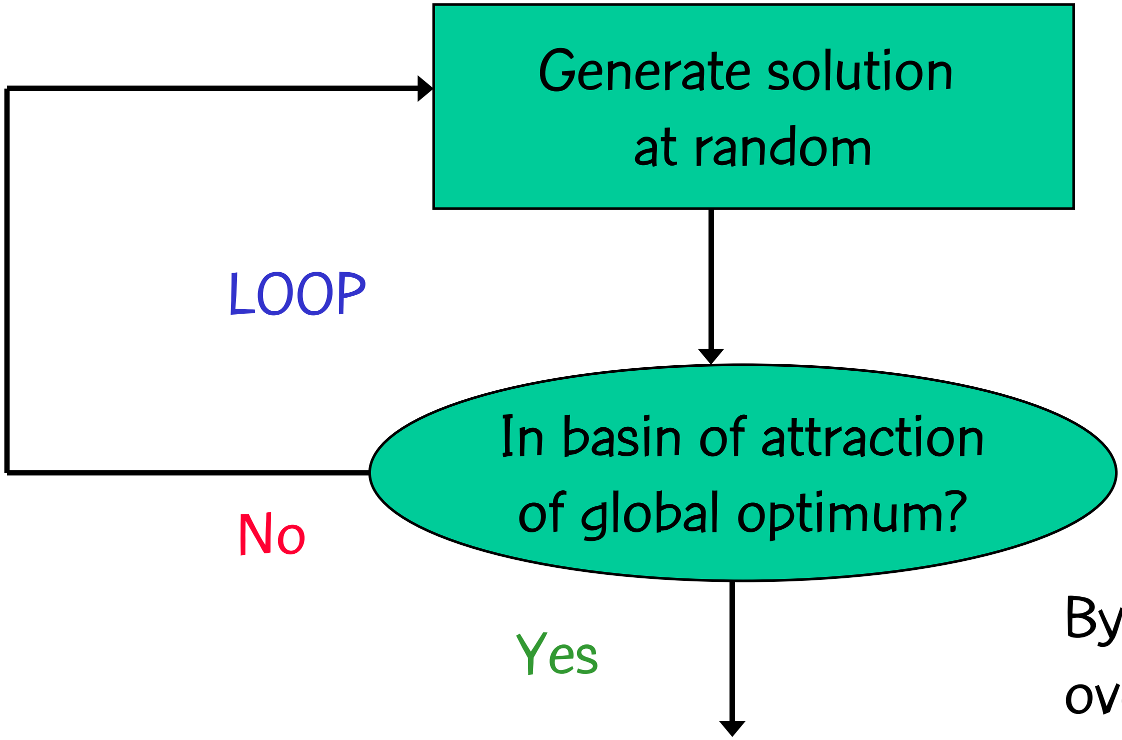
Example: probability of finding opt by random selection

- Suppose x = (0/1, 0/1, 0/1, 0/1, 0/1) and let the unique optimum be x* = (1,0,0,1,1).
- The prob of finding the opt at random is 1/32 = .031 and the prob of not finding it is 31/32.
- After k trials, the probability of not finding the opt is $(31/32)^k$ and hence the prob of find it at least once is $1 - (31/32)^k$
- For k = 5, p = .146; for k = 10, p = .272; for k = 20, p = .470; for k = 50, p = .796; for k = 100, p = .958; for k = 200, p = .998

Example: Probability of finding opt with K samplings on a 0–1 vector of size N

K: \ N:	10	15	20	25	30
10	.010	.000	.000	.000	.000
100	.093	.003	.000	.000	.000
1000	.624	.030	.000	.000	.000
10000	1.000	.263	.009	.000	.000
100000	1.000	.953	.091	.003	.000

Local search with random starting solutions


Generate solution at random
LOOP
In basin of attraction of global optimum?
No
Yes

By repeating LOOP over and over, w.p. 1 outcome is Yes

Local search leads to global optimum.

Greedy algorithm

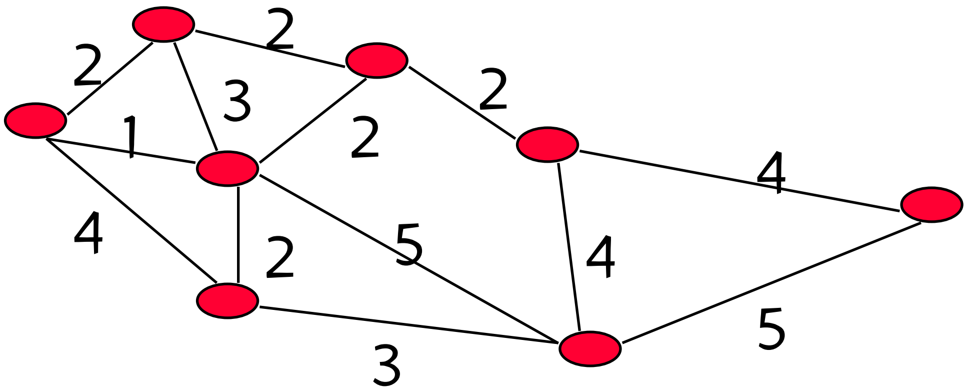
The greedy algorithm

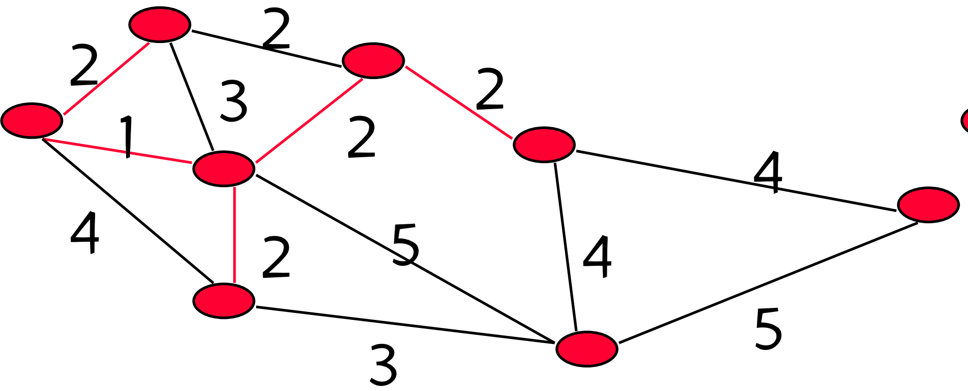
- Constructs a solution, one element at a time:
  - Defines candidate elements.
  - Applies a greedy function to each candidate element.
  - Ranks elements according to greedy function value.
  - Add best ranked element to solution.

repeat until done

The greedy algorithm

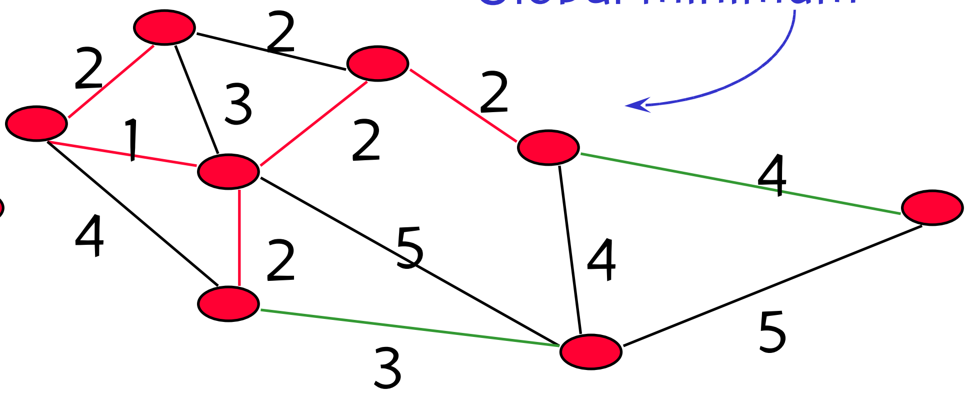
An example: minimum weight spanning tree





Edges of weight 1 & 2

Global minimum



Edges of weight 3 & 4

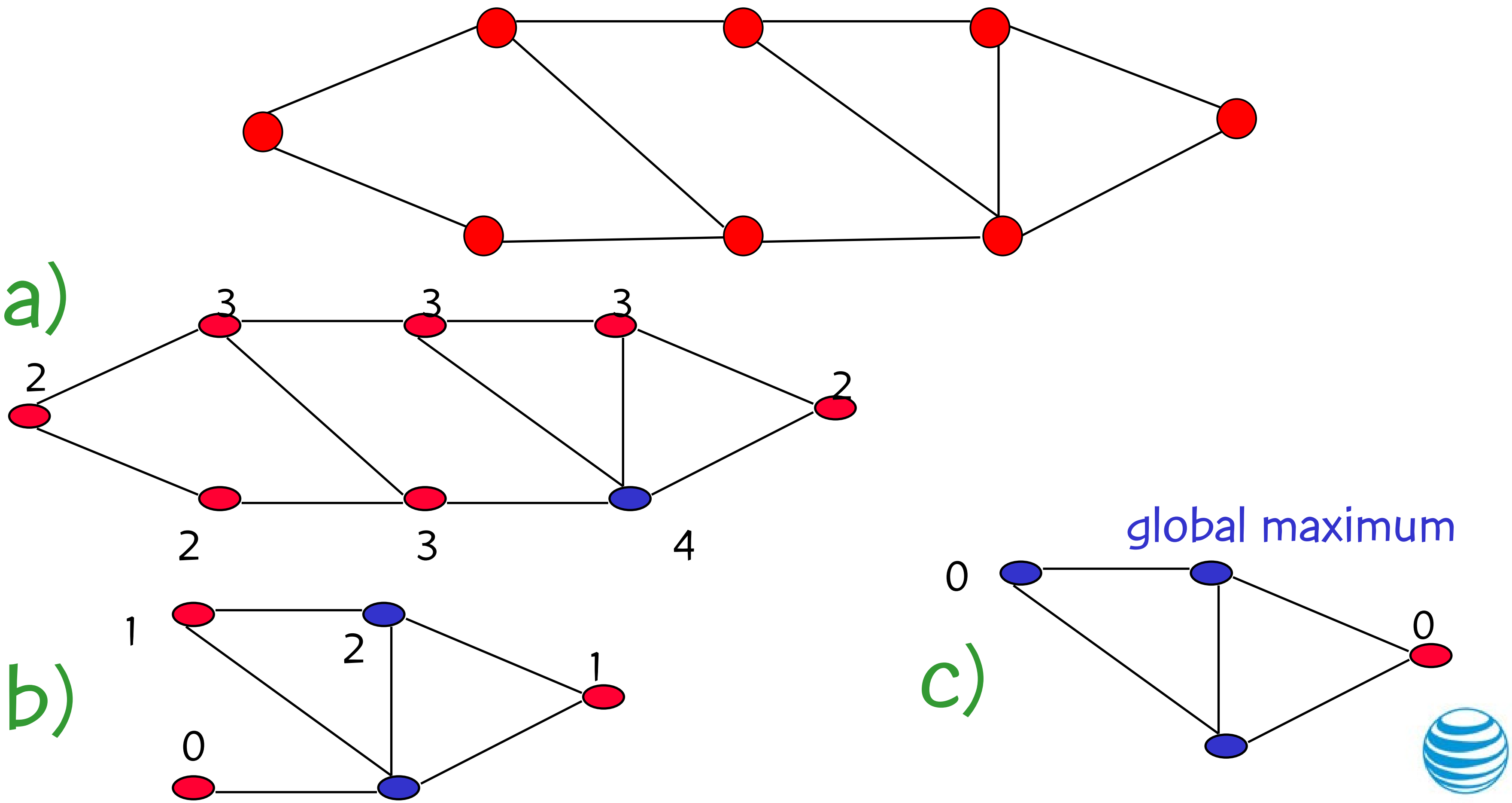
The greedy algorithm

Another example: Maximum clique

- Given graph G = (V, E), find largest subgraph of G such that all vertices are mutually adjacent.
  - greedy algorithm builds solution, one element (vertex) at a time
  - candidate set: unselected vertices adjacent to all selected vertices
  - greedy function: vertex degree with respect to other candidate set vertices.

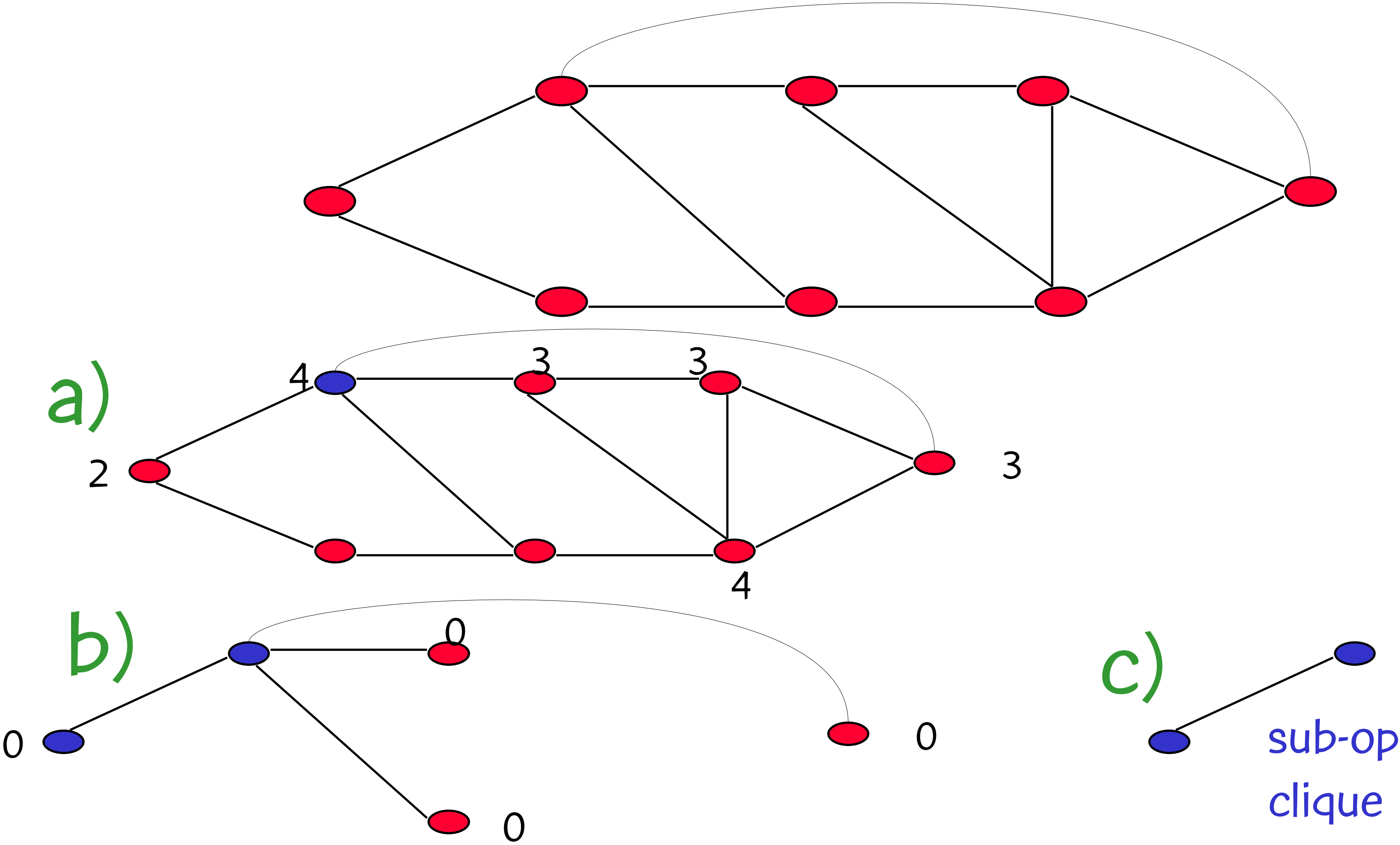
The greedy algorithm

Another example: Maximum clique



The greedy algorithm

Another example: Maximum clique



sub-optimal clique

Semi-greedy heuristic

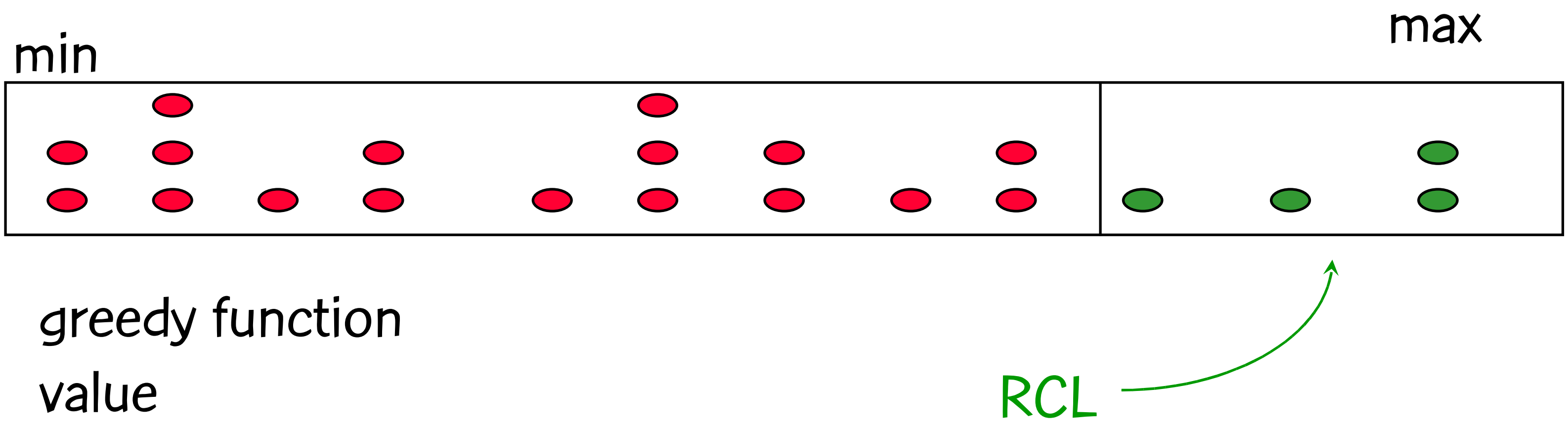
- A semi-greedy heuristic tries to get around convergence to non-global local minima.
- repeat until solution is constructed

repeat until done

  - For each candidate element
    - apply a greedy function to element
  - Rank all elements according to their greedy function values
  - Place well-ranked elements in a restricted candidate list (RCL)
  - Select an element from the RCL at random & add it to the solution

Semi-greedy heuristic

Candidate elements are ranked according to greedy function value.



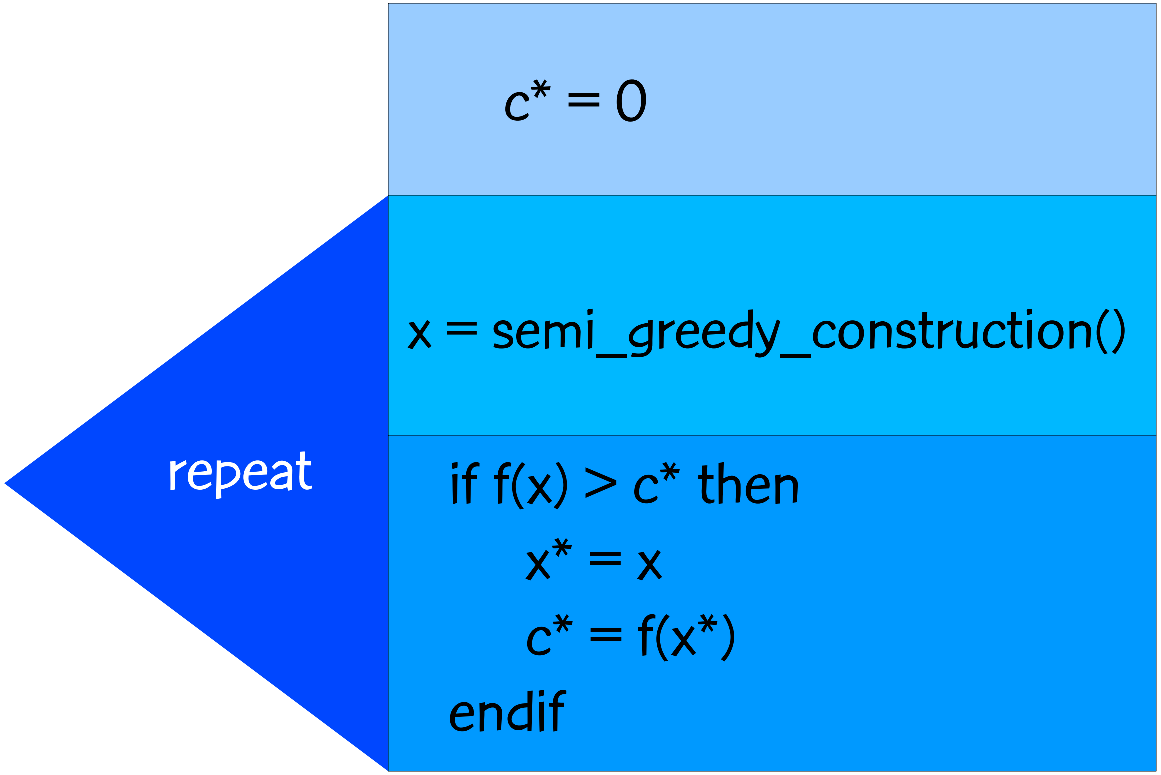
RCL is a set of well-ranked candidate elements.

Semi-greedy heuristic

- Hart & Shogan (1987) propose two mechanisms for building the RCL:
  - Cardinality based: place k best candidates in RCL
  - Value based: place all candidates having greedy values better than $\alpha \cdot$best_value in RCL, where $\alpha \in [0,1]$.
- Feo & Resende (1989) proposed semi-greedy construction, independently, as a basic component of GRASP.

Hart-Shogan Algorithm

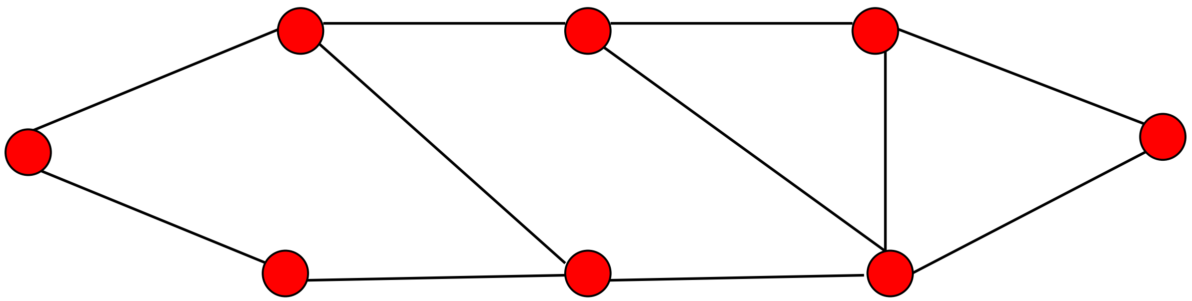
(for maximization problem)

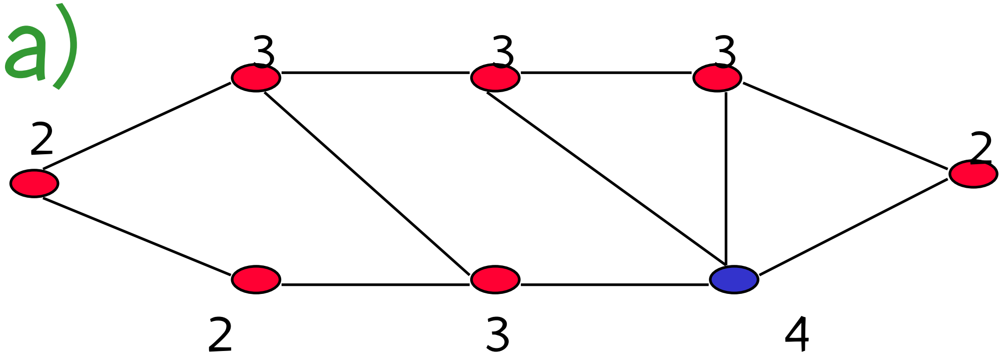


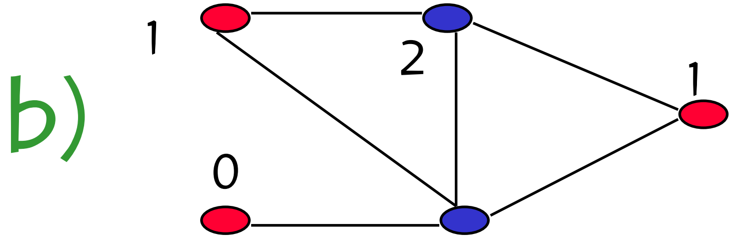
```
c* = 0
repeat
    x = semi_greedy_construction()
    if f(x) > c* then
        x* = x
        c* = f(x*)
    endif
```

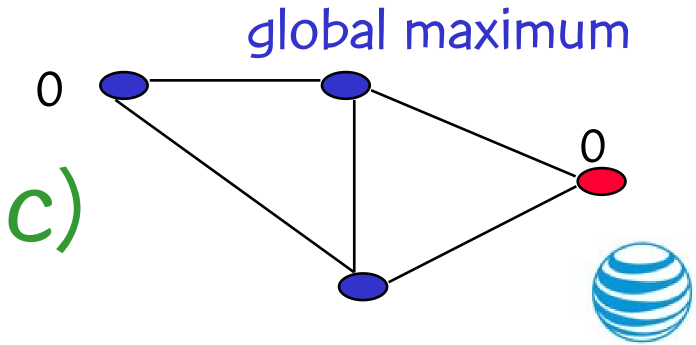
The semi-greedy algorithm

Maximum clique example



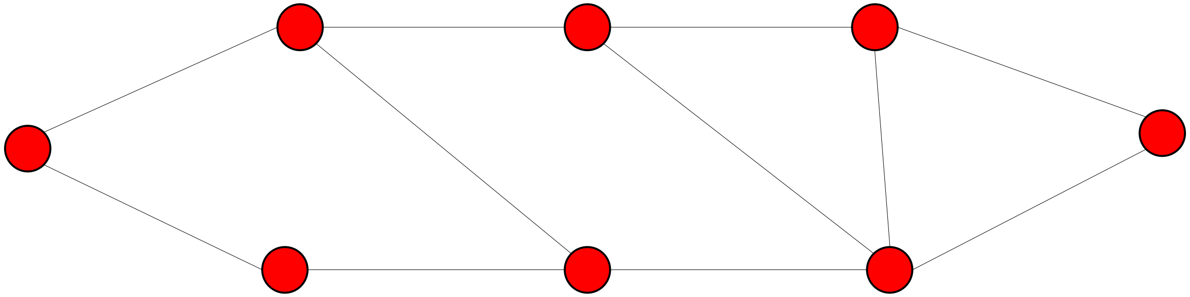






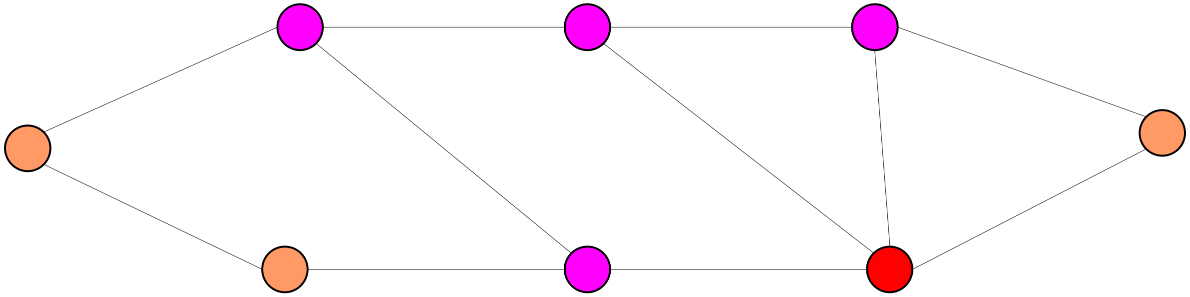
The semi-greedy algorithm

Maximum clique example



The semi-greedy algorithm

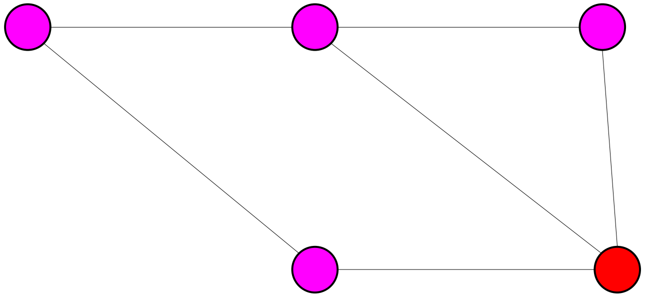
Maximum clique example



Build clique, one node at a time.

Candidates: nodes adjacent to clique.

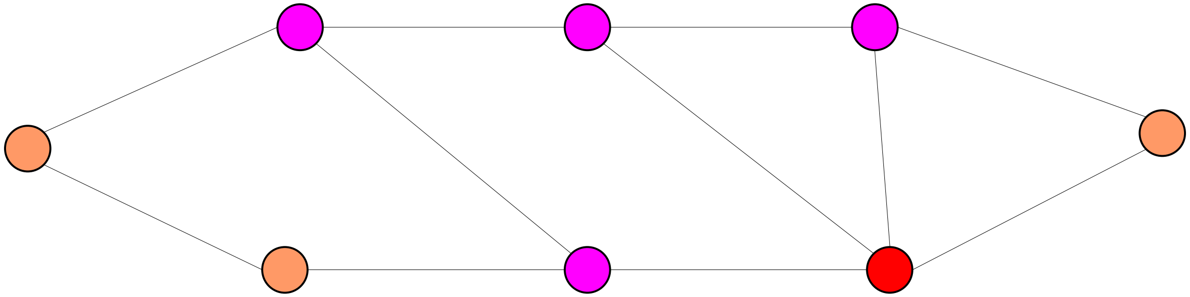
Greedy function: degree with respect to candidate nodes.



RCL =

The semi-greedy algorithm

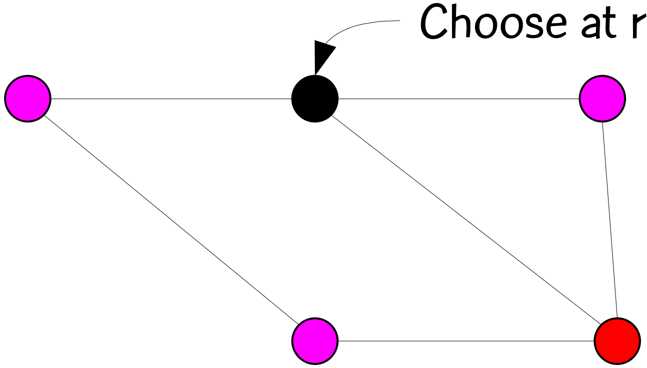
Maximum clique example



Build clique, one node at a time.

Candidates: nodes adjacent to clique.

Greedy function: degree with respect to candidate nodes.



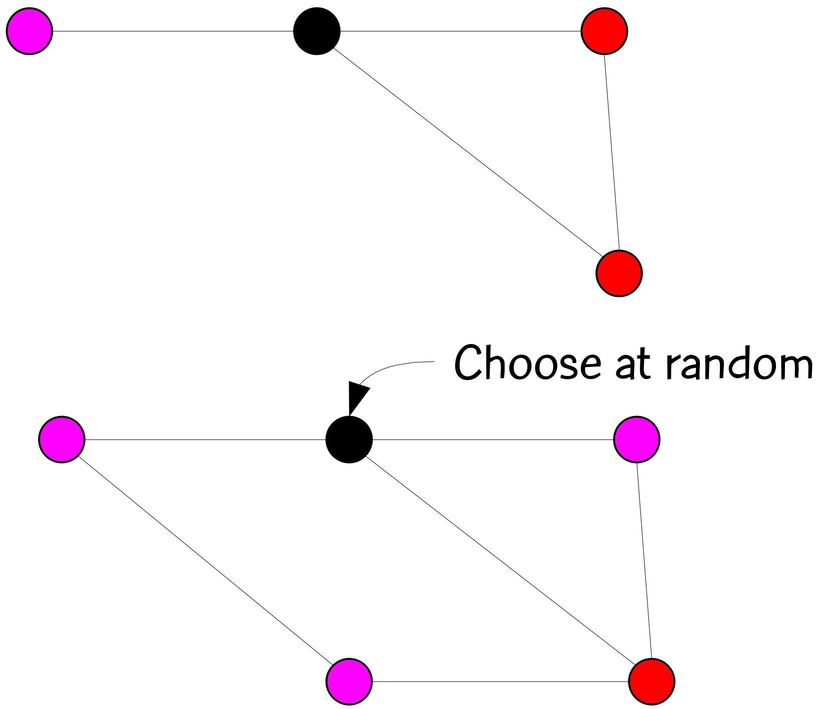
Choose at random

RCL =

Semi-greedy iteration 1

The semi-greedy algorithm

Maximum clique example



Build clique, one node at a time.

Candidates: nodes adjacent to clique.

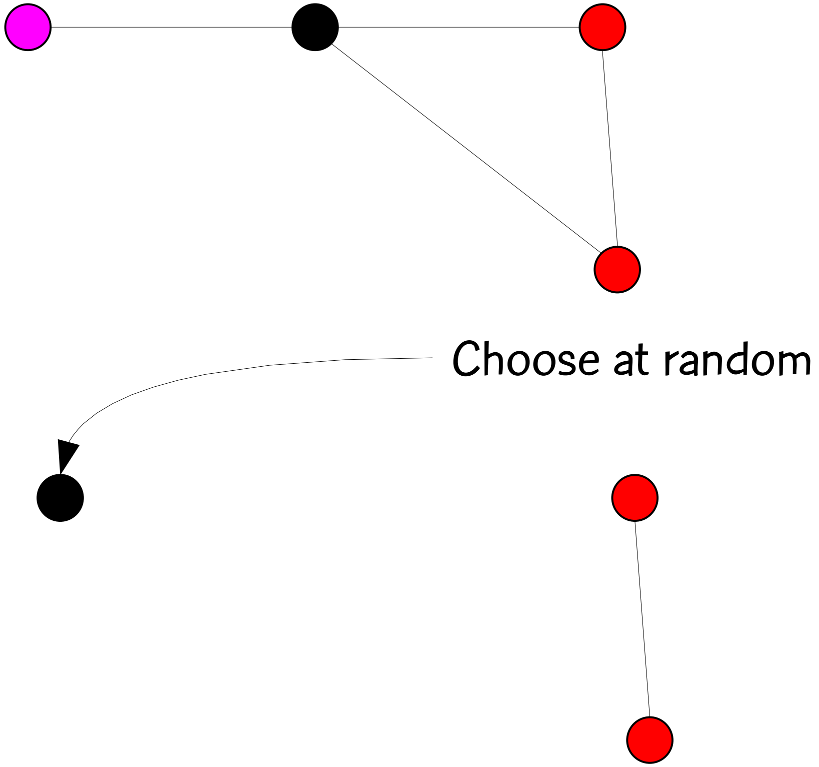
Greedy function: degree with respect to candidate nodes.

RCL =

Semi-greedy iteration 1

The semi-greedy algorithm

Maximum clique example



Choose at random

RCL =

Build clique, one node at a time.

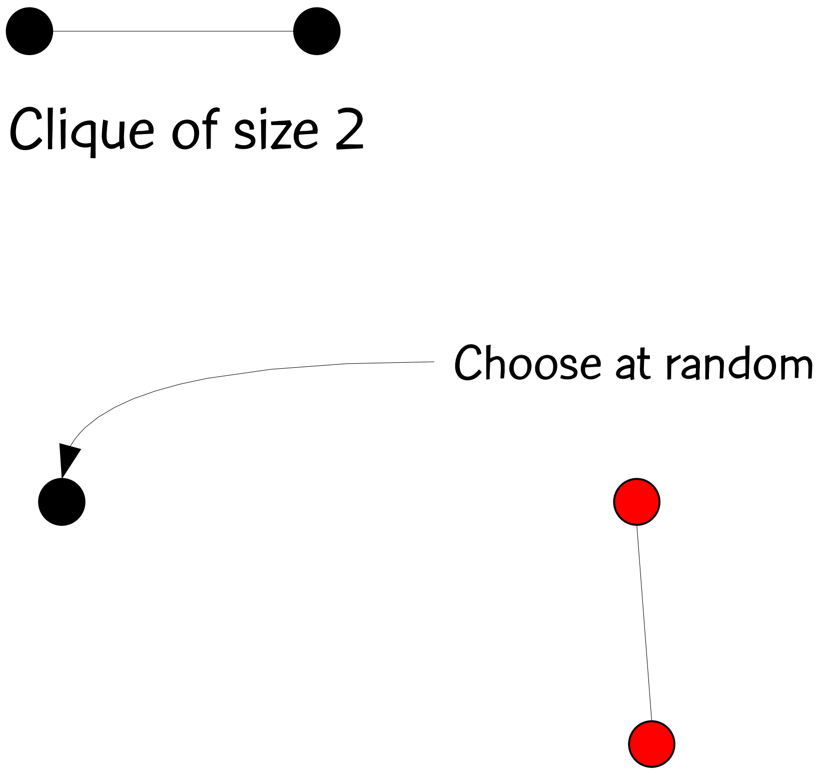
Candidates: nodes adjacent to clique.

Greedy function: degree with respect to candidate nodes.

Semi-greedy iteration 1

The semi-greedy algorithm

Maximum clique example



Clique of size 2

Choose at random

RCL =

Build clique, one node at a time.

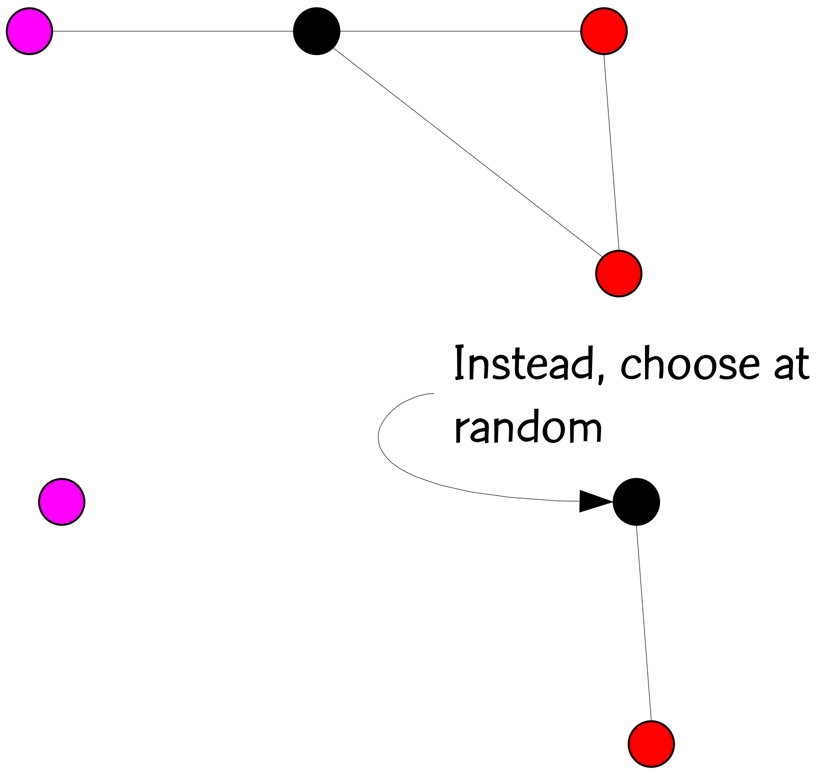
Candidates: nodes adjacent to clique.

Greedy function: degree with respect to candidate nodes.

Semi-greedy iteration 1

The semi-greedy algorithm

Maximum clique example



Instead, choose at random

Build clique, one node at a time.

Candidates: nodes adjacent to clique.

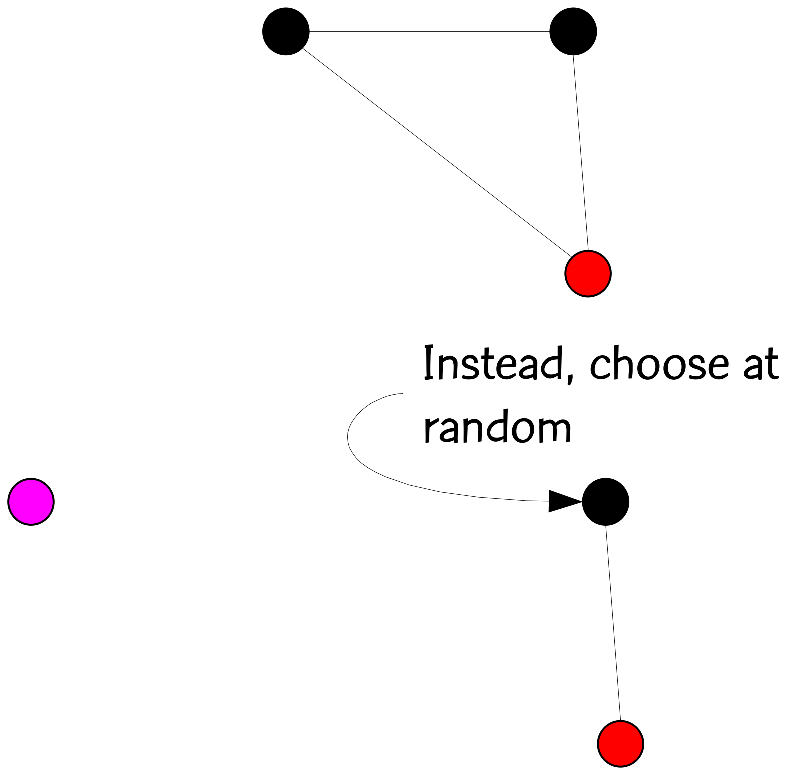
Greedy function: degree with respect to candidate nodes.

RCL =

Semi-greedy iteration 2

The semi-greedy algorithm

Maximum clique example



Build clique, one node at a time.

Candidates: nodes adjacent to clique.

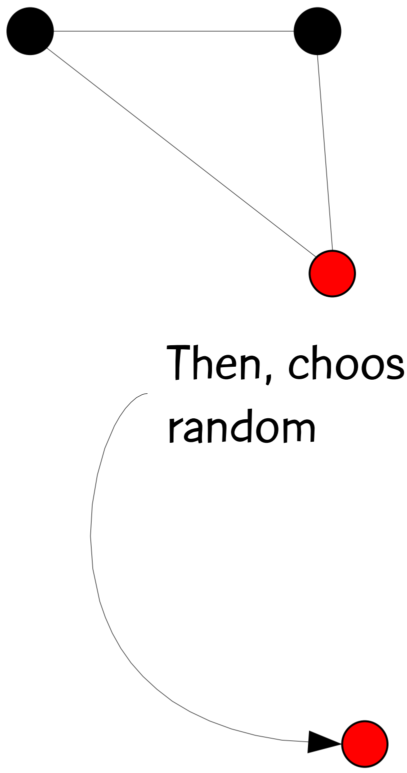
Greedy function: degree with respect to candidate nodes.

RCL =

Semi-greedy iteration 2

The semi-greedy algorithm

Maximum clique example



Then, choose at random

Build clique, one node at a time.

Candidates: nodes adjacent to clique.

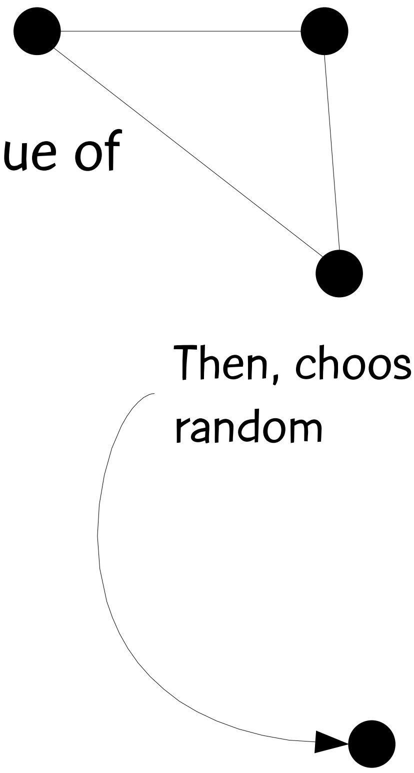
Greedy function: degree with respect to candidate nodes.

RCL =

Semi-greedy iteration 2

The semi-greedy algorithm

Maximum clique example



Optimal clique of size 3

Then, choose at random

RCL =

Build clique, one node at a time.

Candidates: nodes adjacent to clique.

Greedy function: degree with respect to candidate nodes.

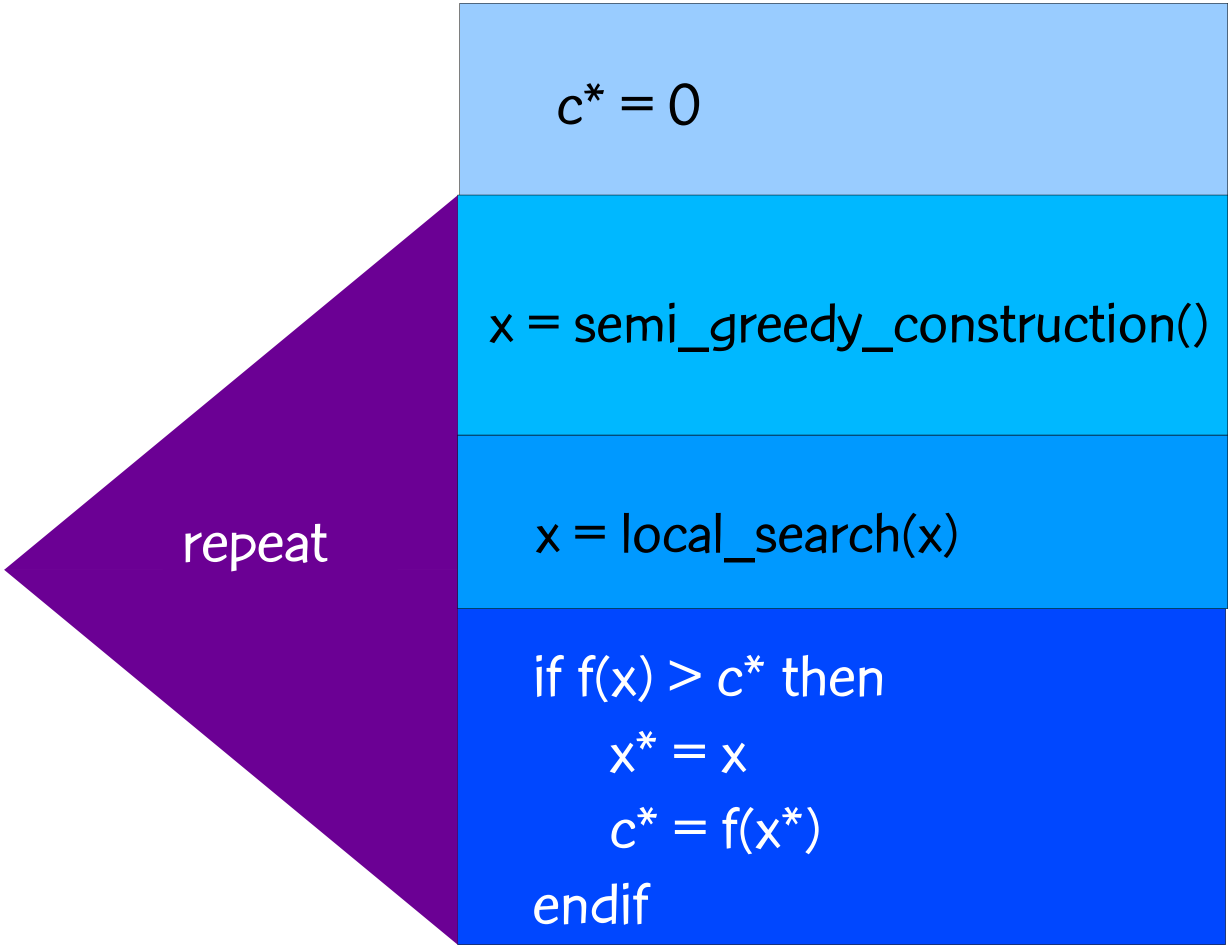
Semi-greedy iteration 2

GRASP

GRASP: Basic algorithm

- GRASP is a multistart metaheuristic. Some references:
    - Feo & Resende (1989): GRASP introduced for set covering
    - Resende (1989): talk on GRASP at INFORMS NYC Meeting
    - Feo & Resende (1991): tutorial at INFORMS Nashville Meeting
    - Feo & Resende (1995): first survey
    - Festa & Resende (2002): annotated bibliography
    - Resende & Ribeiro (2003): most recent survey
    - Resende & González Velarde (2003): survey in Spanish

GRASP: Basic algorithm



```
$c^* = 0$
repeat
    $x$ = semi_greedy_construction()
    $x$ = local_search($x$)
    if $f(x) > c^*$ then
        $x^* = x$
        $c^* = f(x^*)$
    endif
```

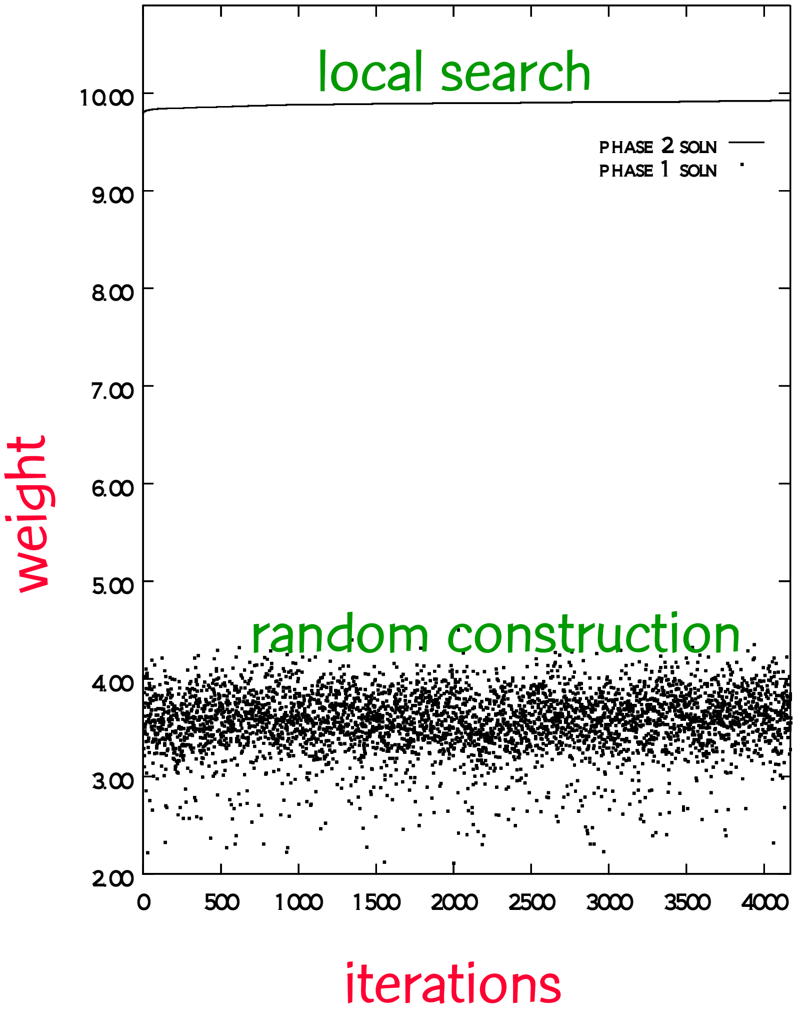
Semi-greediness is more general in GRASP

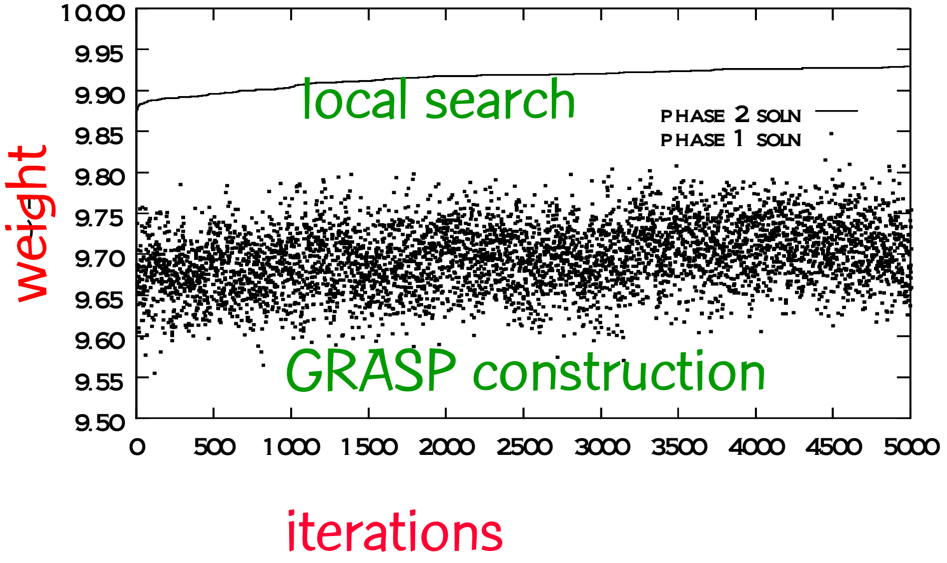
GRASP: Basic algorithm

- Construction phase: greediness + randomization
  - Builds a feasible solution combining greediness and randomization
- Local search: search in the current neighborhood until a local optimum is found
  - Solutions generated by the construction procedure are not necessarily optimal:
    - Effectiveness of local search depends on: neighborhood structure, search strategy, and fast evaluation of neighbors, but also on the construction procedure itself.

GRASP Construction

GRASP: Basic algorithm





Effectiveness of greedy randomized vs purely randomized construction:

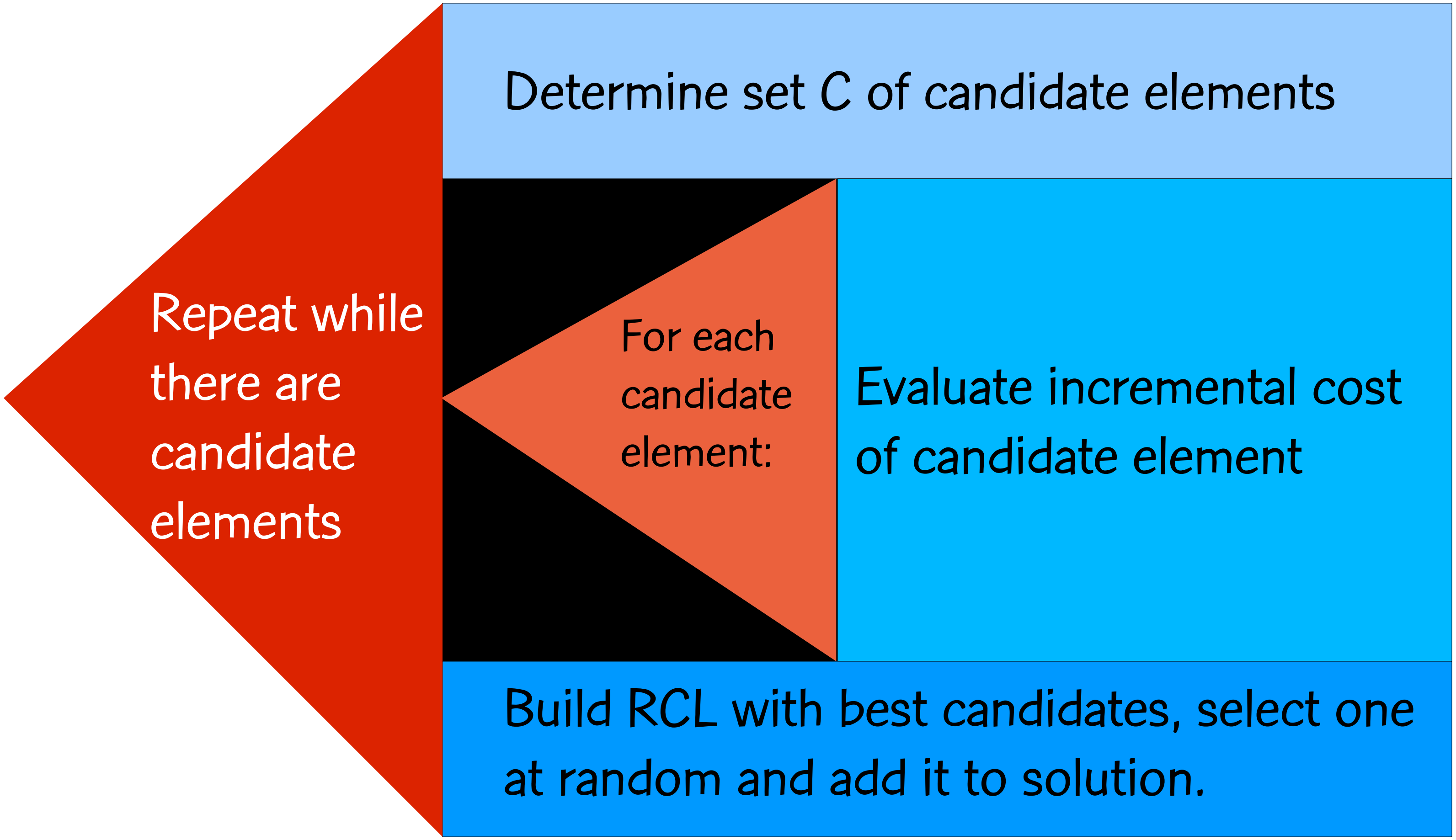
Application: modem placement

max weighted covering problem

maximization problem: $\alpha = 0.85$

Construction phase: RCL based

restricted candidate list



Determine set C of candidate elements

Repeat while there are candidate elements

For each candidate element:

Evaluate incremental cost of candidate element

Build RCL with best candidates, select one at random and add it to solution.

Construction phase: RCL based

- Minimization problem
- Basic construction procedure:
  - Greedy function *c(e)*: incremental cost associated with the incorporation of element e into the current partial solution under construction
  - c^{min} (resp. c^{max}): smallest (resp. largest) incremental cost
  - RCL made up by the elements with the smallest incremental costs.

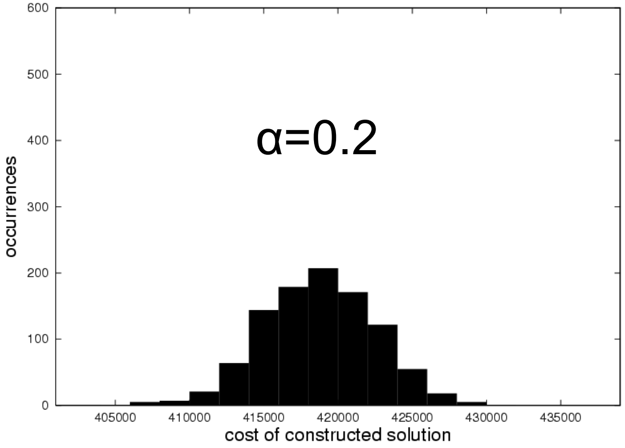
Construction phase

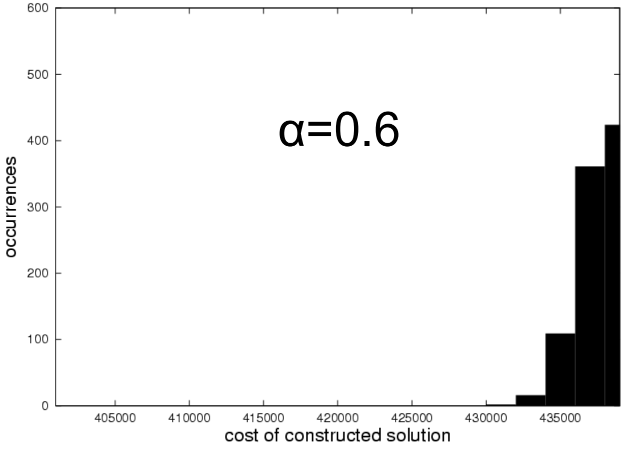
- Cardinality-based construction:
  - p elements with the smallest incremental costs
- Quality-based construction:
  - Parameter α defines the quality of the elements in RCL.
  - RCL contains elements with incremental cost

    $$c^{\min} \leq c(e) \leq c^{\min} + \alpha\,(c^{\max} - c^{\min})$$

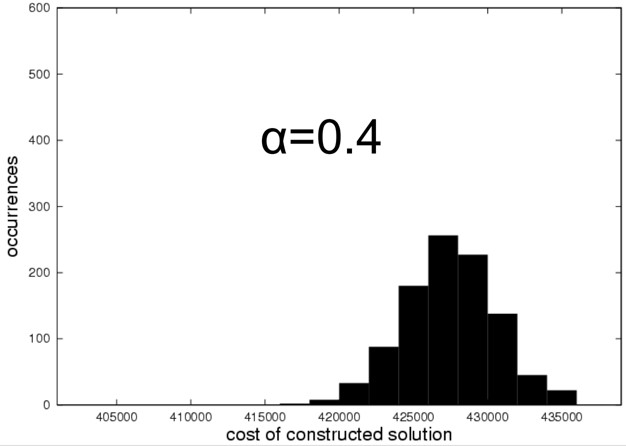
    - $\alpha = 0$: pure greedy construction
    - $\alpha = 1$: pure randomized construction
- Select at random from RCL using uniform probability distribution

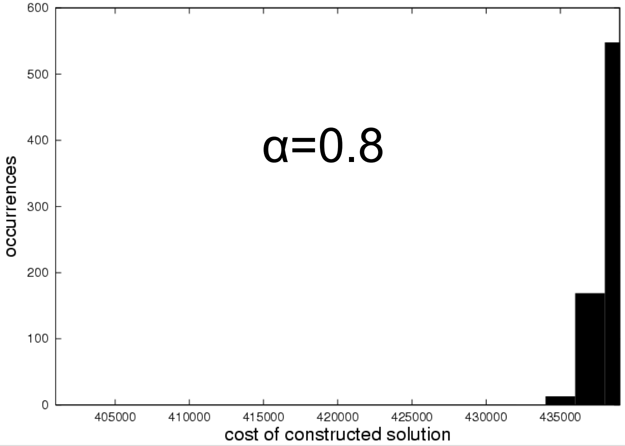
Illustrative results: RCL parameter





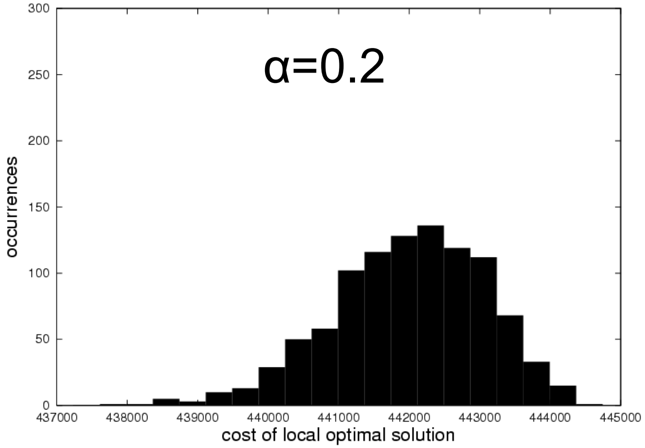
Construction phase only

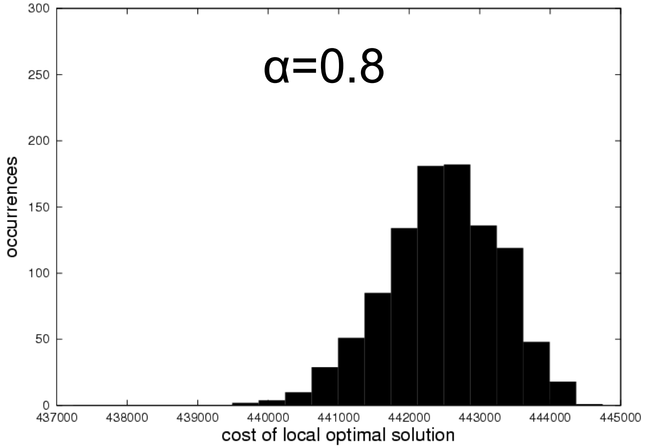




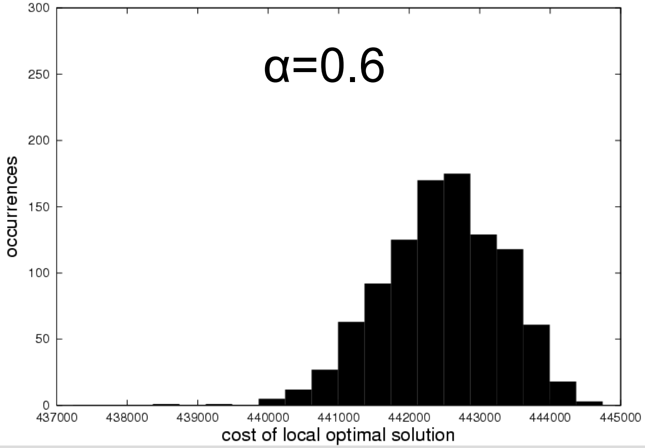
weighted MAX-SAT instance, 1000 GRASP iterations

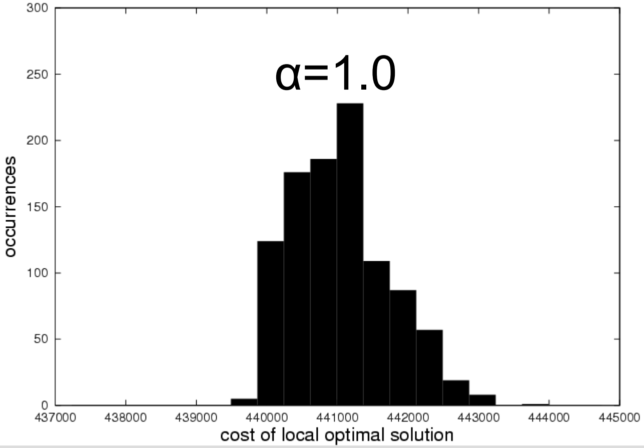
Illustrative results: RCL parameter





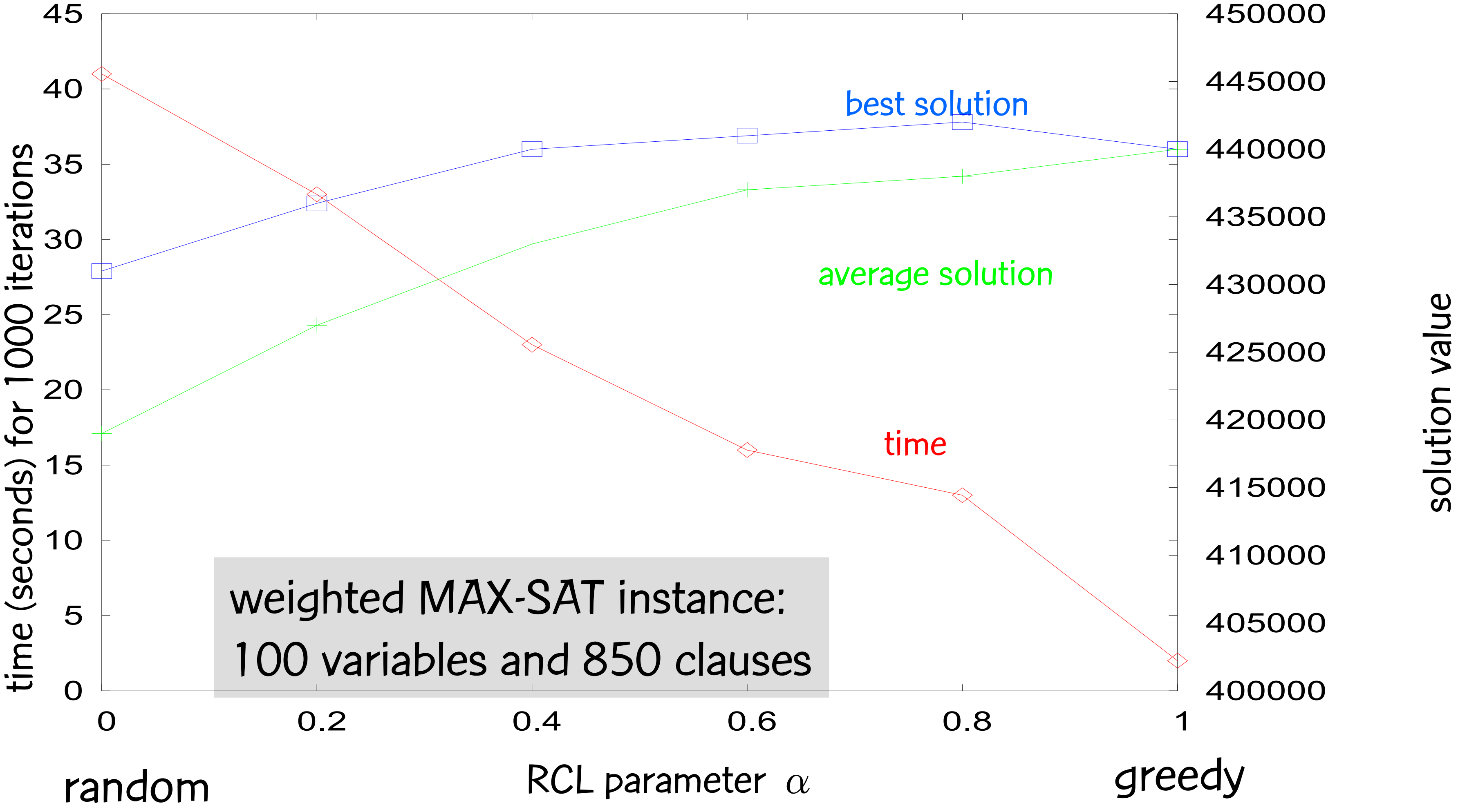
Construction + local search





weighted MAX-SAT instance, 1000 GRASP iterations

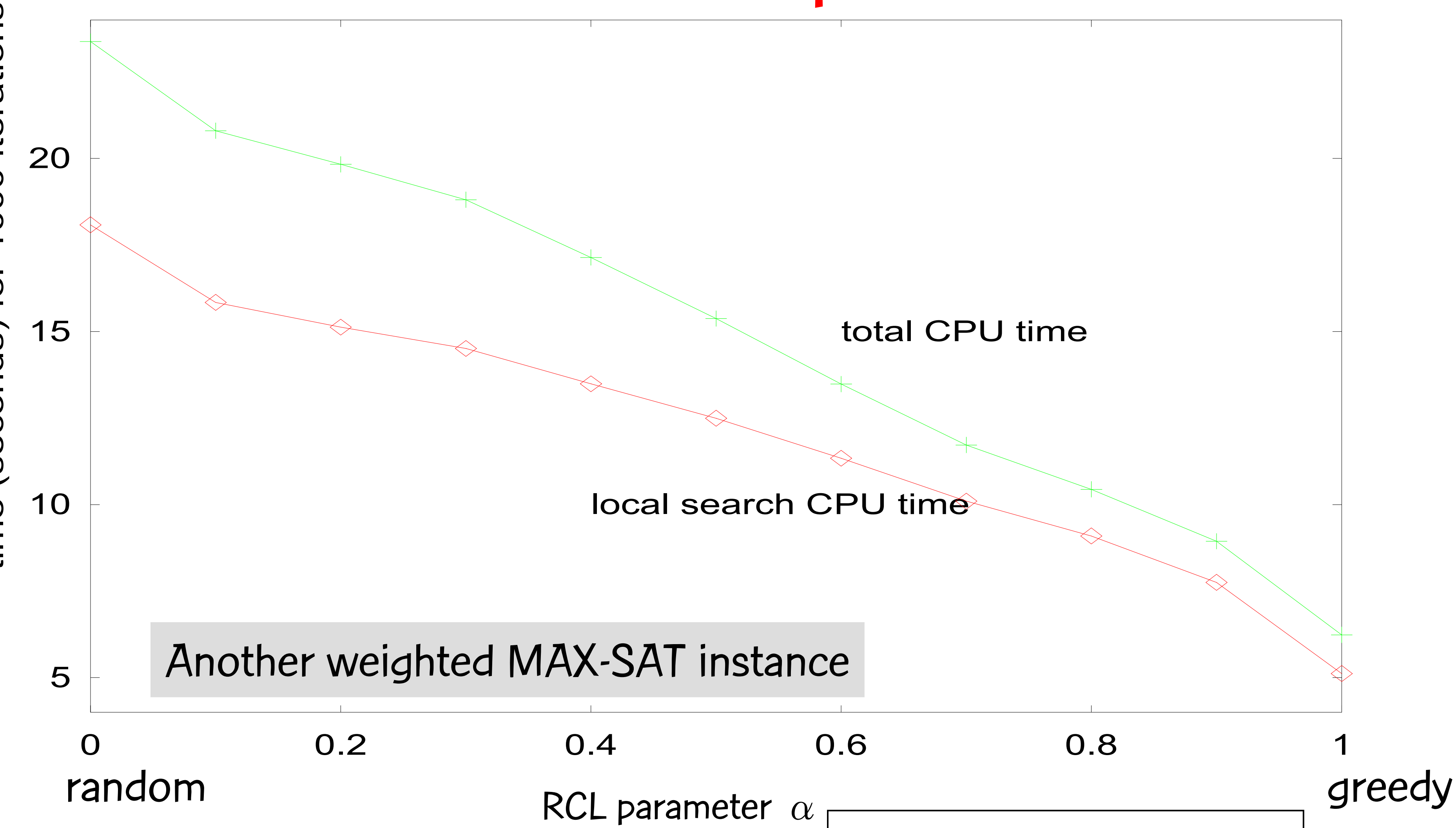
Illustrative results: RCL parameter



weighted MAX-SAT instance:
100 variables and 850 clauses

SGI Challenge 196 MHz

Illustrative results: RCL parameter

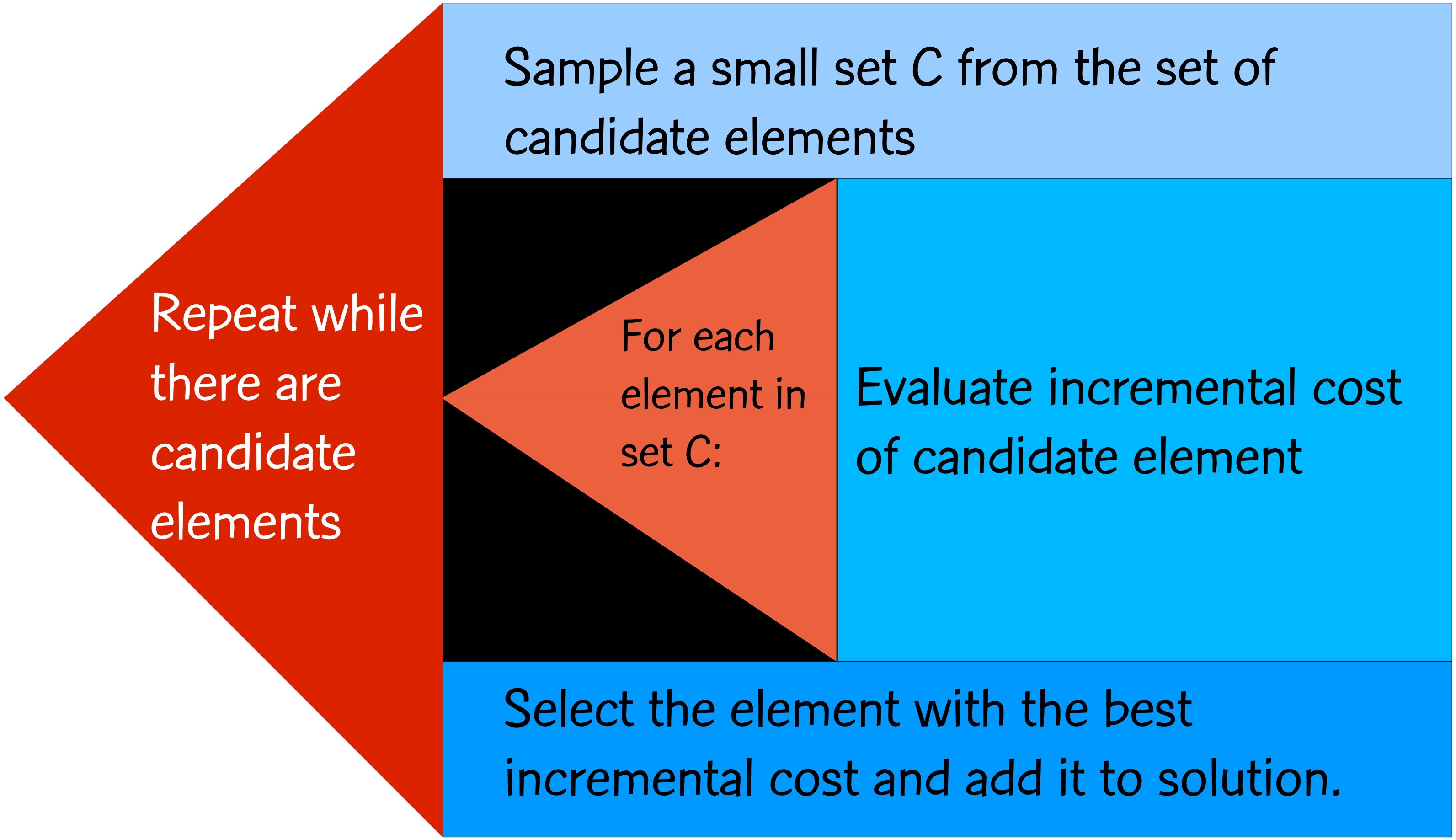


Another weighted MAX-SAT instance

SGI Challenge 196 MHz

Construction phase: sampled greedy

[Resende & Werneck, 2004]



Repeat while there are candidate elements

Sample a small set C from the set of candidate elements

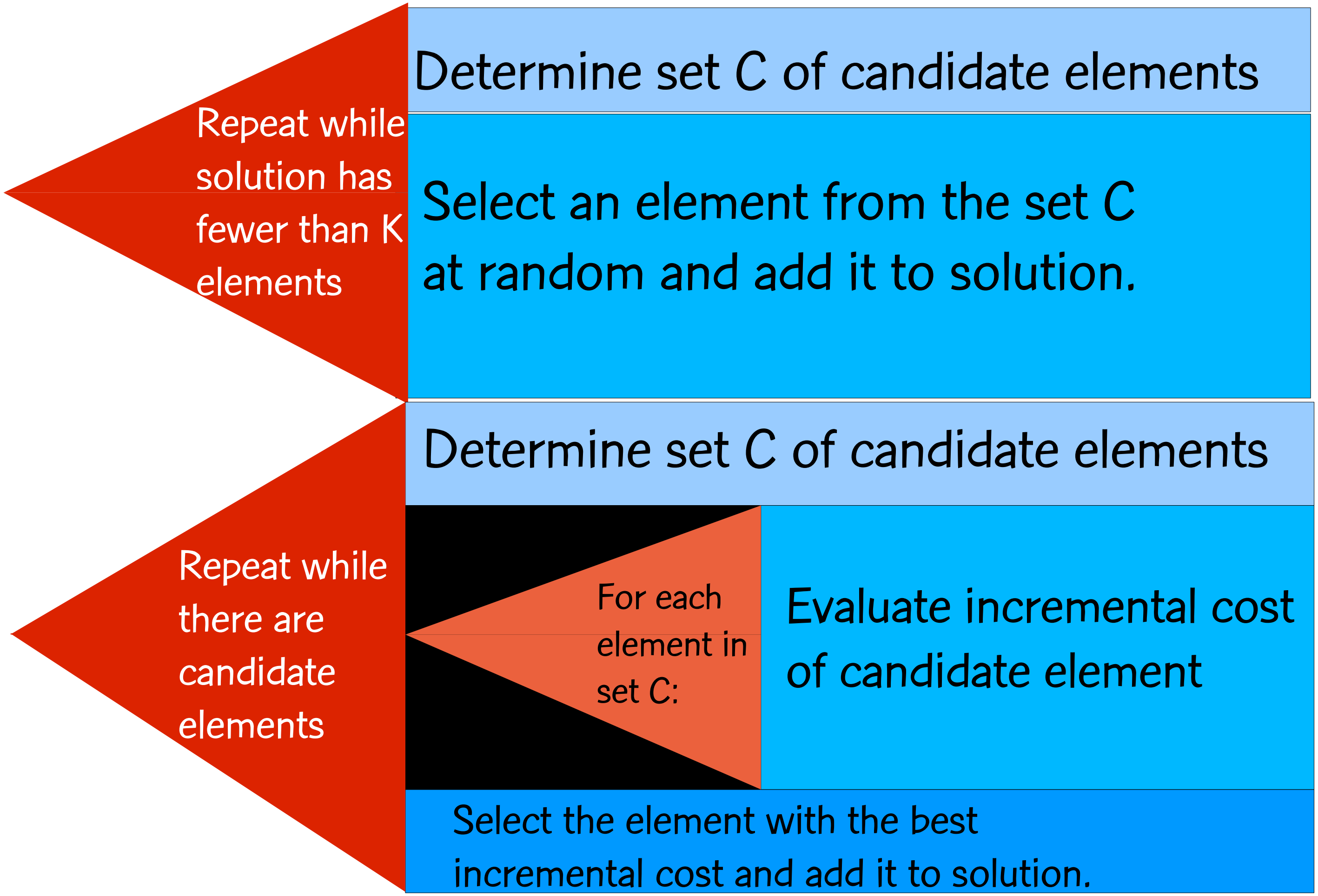
For each element in set C:

Evaluate incremental cost of candidate element

Select the element with the best incremental cost and add it to solution.

Construction phase: random+greedy

[Resende & Werneck, 2004]



Repeat while solution has fewer than K elements

Determine set C of candidate elements

Select an element from the set C at random and add it to solution.

Repeat while there are candidate elements

Determine set C of candidate elements

For each element in set C:

Evaluate incremental cost of candidate element

Select the element with the best incremental cost and add it to solution.

Construction phase: bias function

[Bresina, 1996]

- In RCL scheme, next element is selected at random (uniformly) from elements in RCL.
- Sorts candidates σ by greedy function and assigning a probability $\pi(\sigma)$ of selection proportional to the element's rank $r(\sigma)$.

$\pi(\sigma) = \text{bias}(r(\sigma))/\Sigma\ \{\text{bias}(r(\sigma')) \mid \sigma' \in \text{RCL}\}$

where bias(r) can be of several types:

1) random: $\text{bias}(r) = 1$

2) linear: $\text{bias}(r) = 1/r$

3) log: $\text{bias}(r) = 1/\log(r+1)$

4) exponential: $\text{bias}(r) = e^{-r}$

5) n-polynomial: $\text{bias}(r) = r^{-n}$

Construction phase: bias function

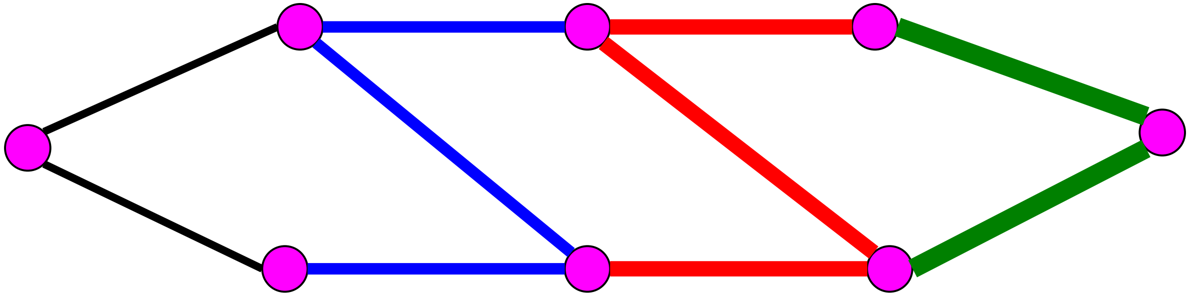
(selection probabilities)

Rank	1	2	3	4	5
Bias function					
Random	0.2	0.2	0.2	0.2	0.2
Linear	0.31	0.15	0.1	0.08	0.06
Log	0.34	0.21	0.17	0.15	0.13
Exp	0.63	0.23	0.09	0.03	0.02
2-polynomial	0.68	0.17	0.08	0.04	0.03

Construction with cost perturbation

- Introduces noise into original costs: similar to Noisy Method of Charon and Hudry (1993, 2002)
- Randomly perturb original costs and apply some heuristic.
- Adds flexibility to algorithm design:
  - May be more effective than greedy randomized construction in circumstances where the construction algorithm is not very sensitive to randomization (Ribeiro, Uchoa, & Werneck, 2002).
  - Also useful when no greedy algorithm is available (Canuto, Resende, & Ribeiro, 2001).

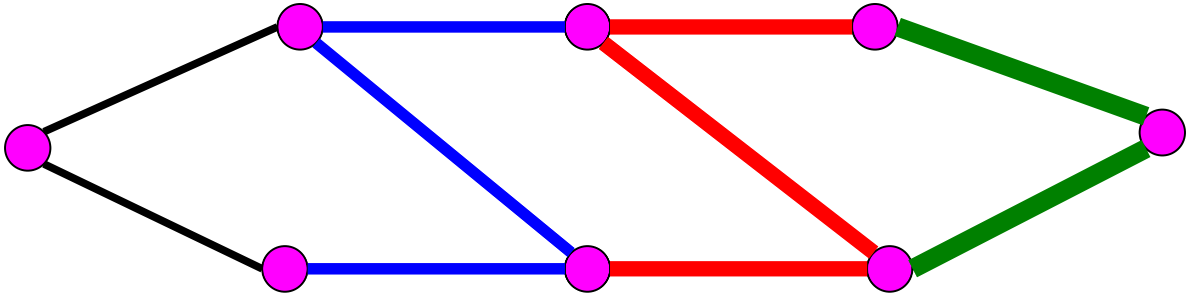
Construction with cost perturbation



W(⬛) < W(🟦) < W(🟥) < W(🟩)

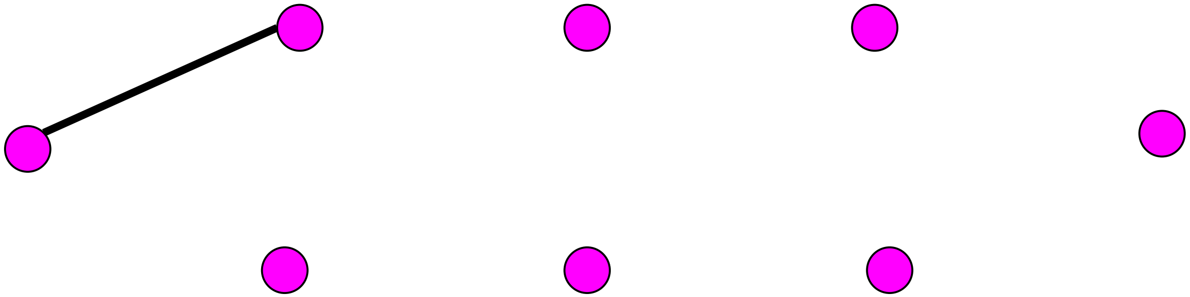
Perturb with costs increasing from top to bottom.

Construction with cost perturbation

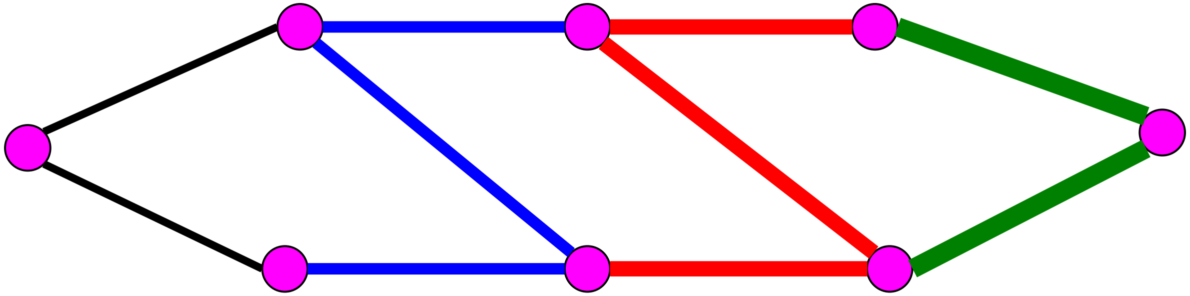


Perturb with costs increasing from top to bottom.

W(⬛) < W(🟦) < W(🟥) < W(🟩)

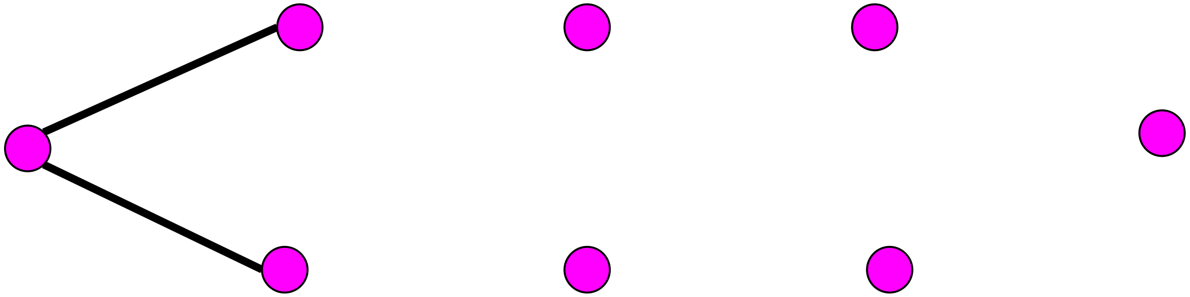


Construction with cost perturbation

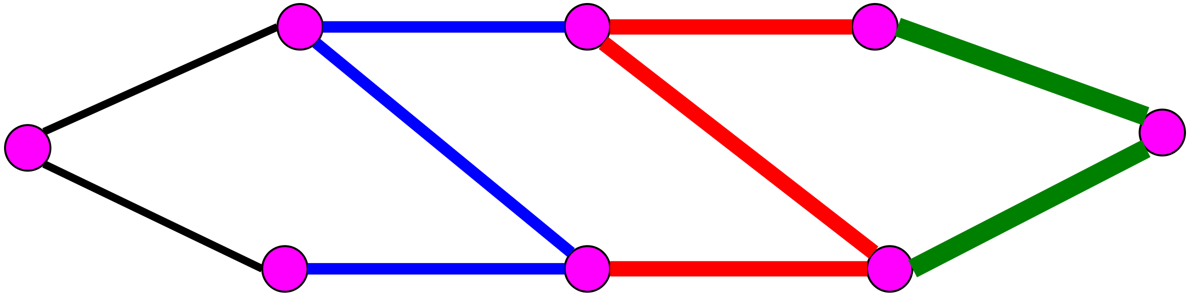


Perturb with costs increasing from top to bottom.

W(⬛) < W(🟦) < W(🟥) < W(🟩)

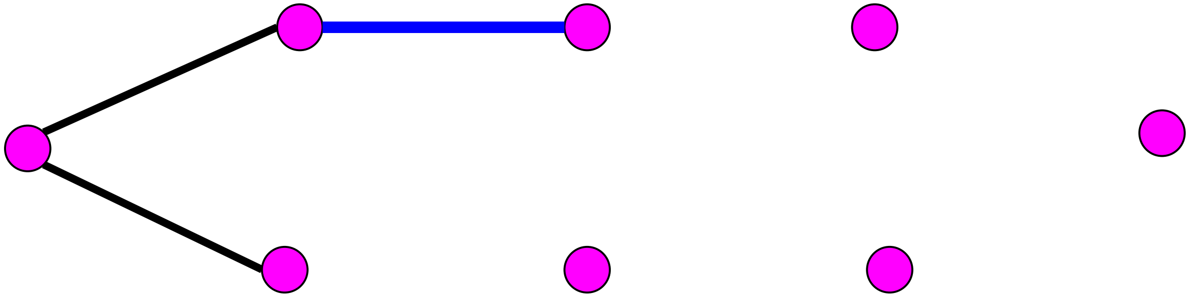


Construction with cost perturbation

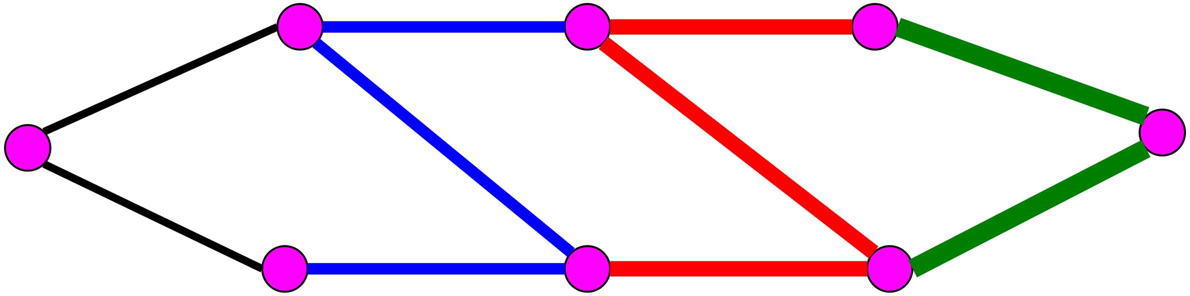


Perturb with costs increasing from top to bottom.

W(⬛) < W(🟦) < W(🟥) < W(🟩)

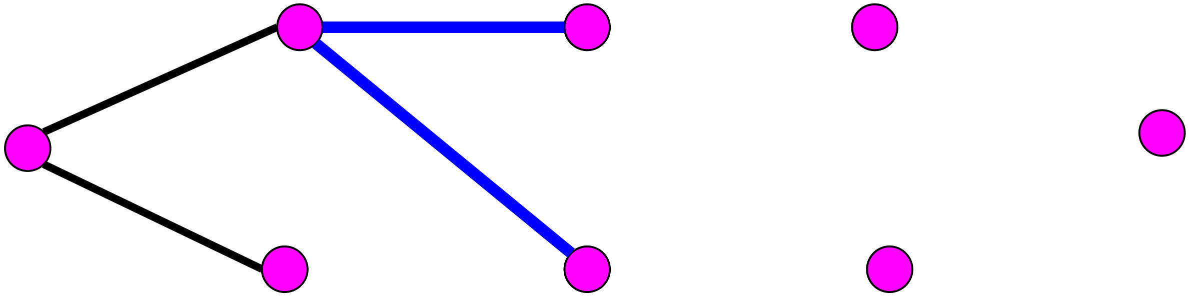


Construction with cost perturbation

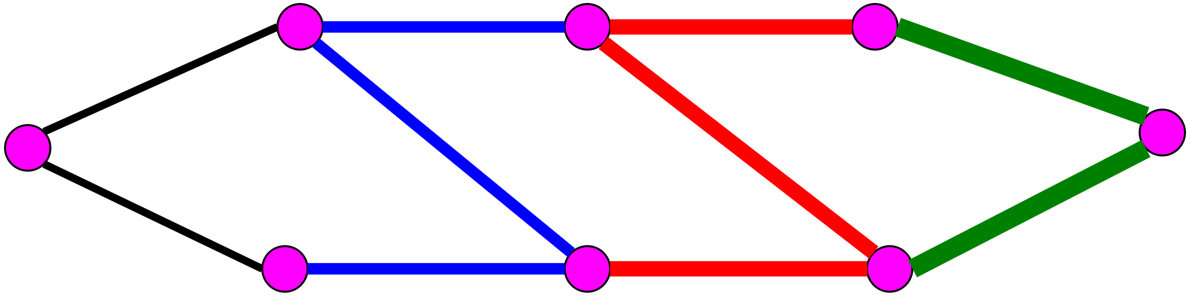


Perturb with costs increasing from top to bottom.

W(⬛) < W(🟦) < W(🟥) < W(🟩)

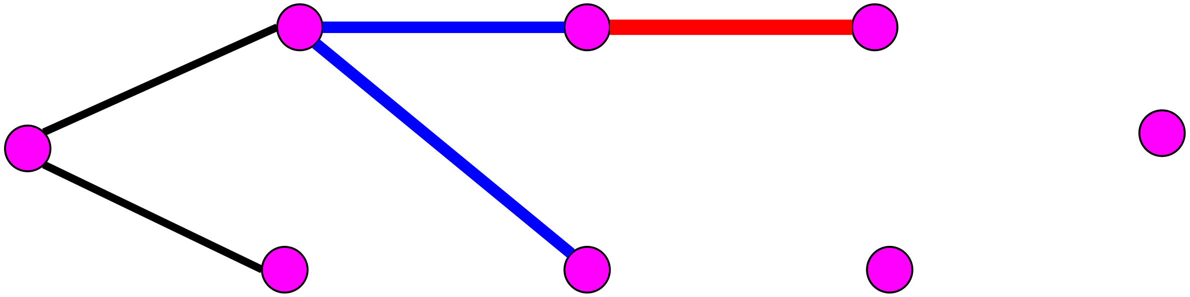


Construction with cost perturbation

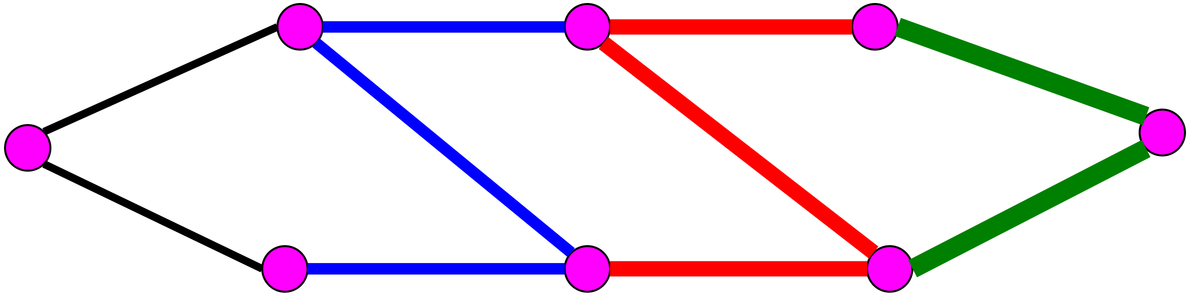


Perturb with costs increasing from top to bottom.

W(⬛) < W(🟦) < W(🟥) < W(🟩)

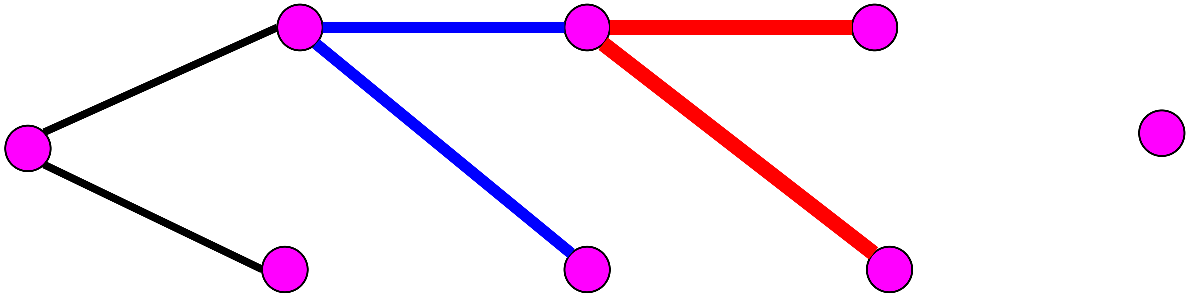


Construction with cost perturbation

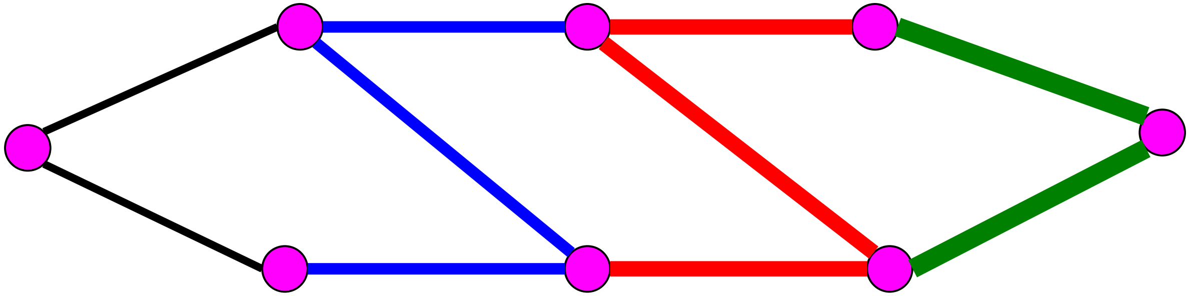


Perturb with costs increasing from top to bottom.

W(⬛) < W(🟦) < W(🟥) < W(🟩)

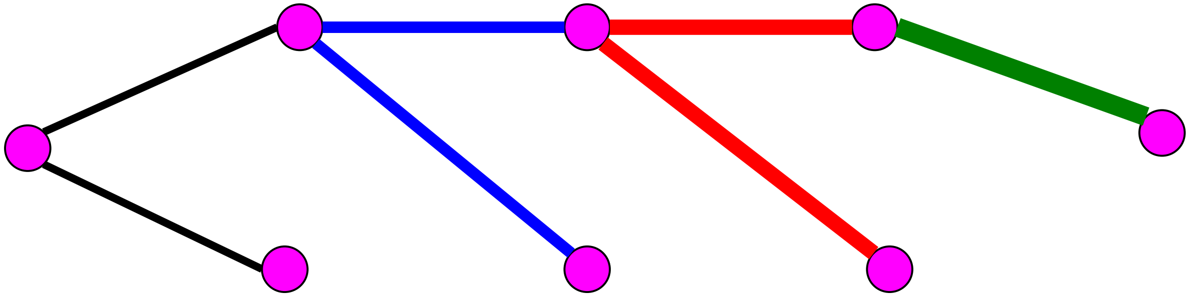


Construction with cost perturbation

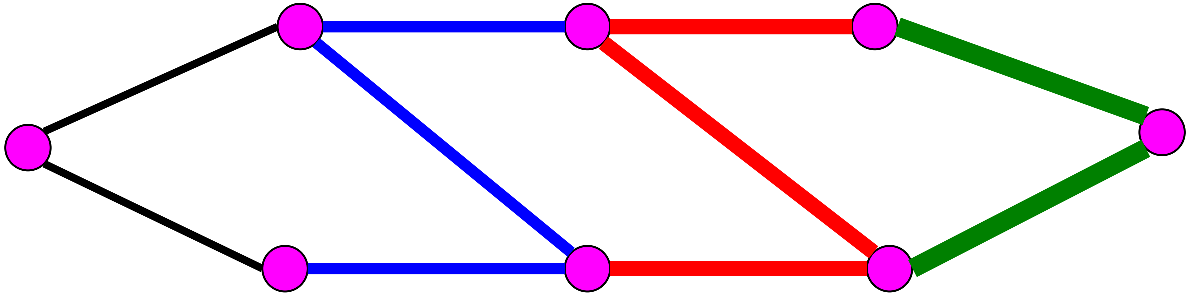


Perturb with costs increasing from top to bottom.

W(⬛) < W(🟦) < W(🟥) < W(🟩)



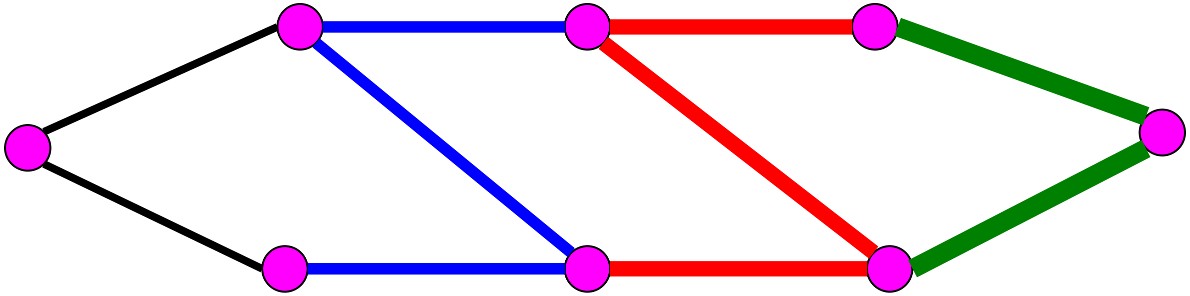
Construction with cost perturbation



W(⬛) < W(🟦) < W(🟥) < W(🟩)

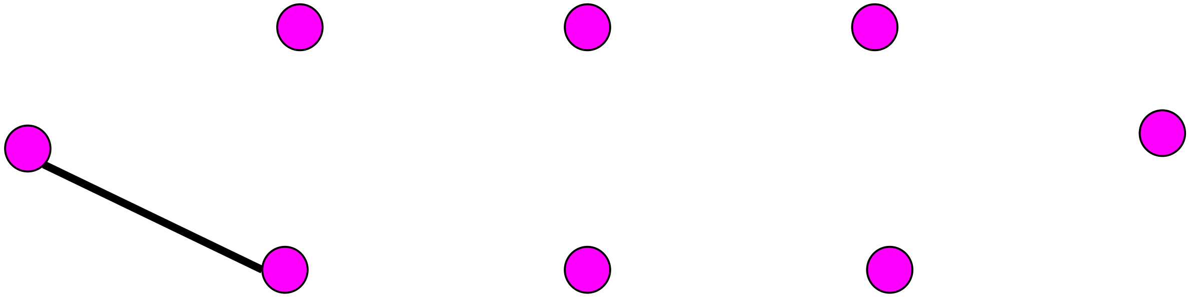
Perturb with costs increasing from bottom to top.

Construction with cost perturbation

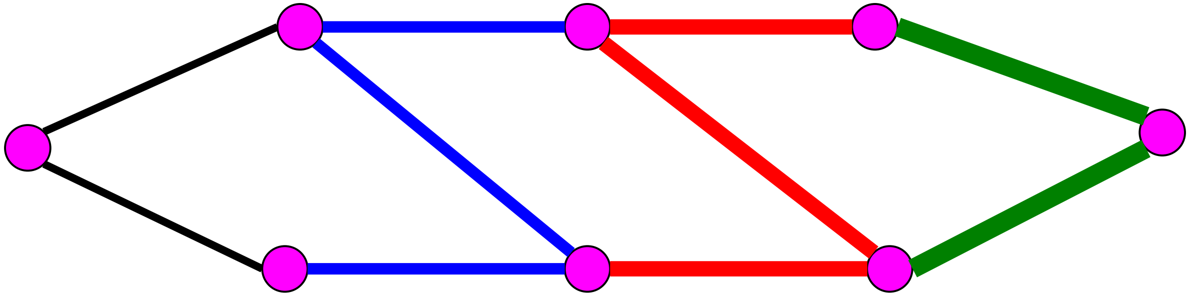


Perturb with costs increasing from bottom to top.

W(■) < W(🟦) < W(🟥) < W(🟩)

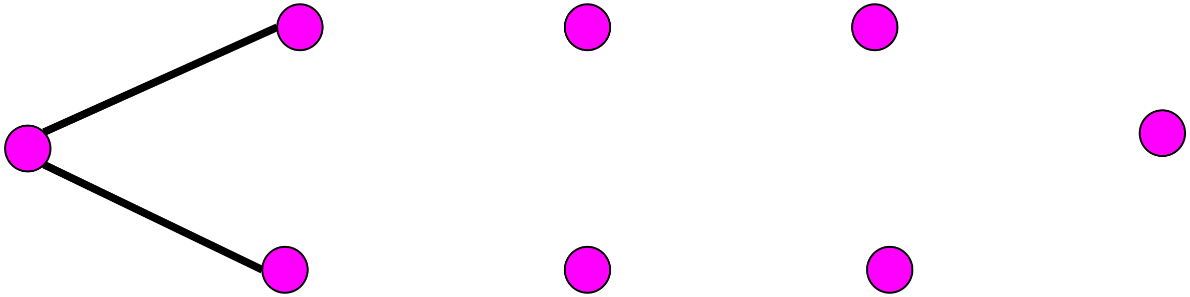


Construction with cost perturbation

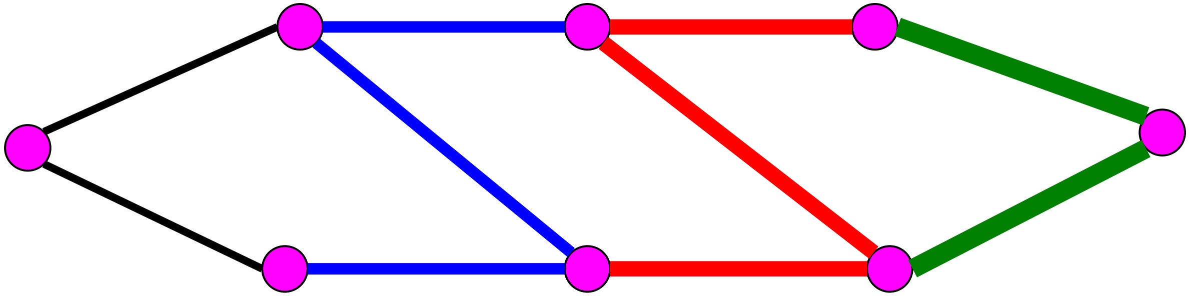


Perturb with costs increasing from bottom to top.

W(▮) < W(🟦) < W(🟥) < W(🟩)

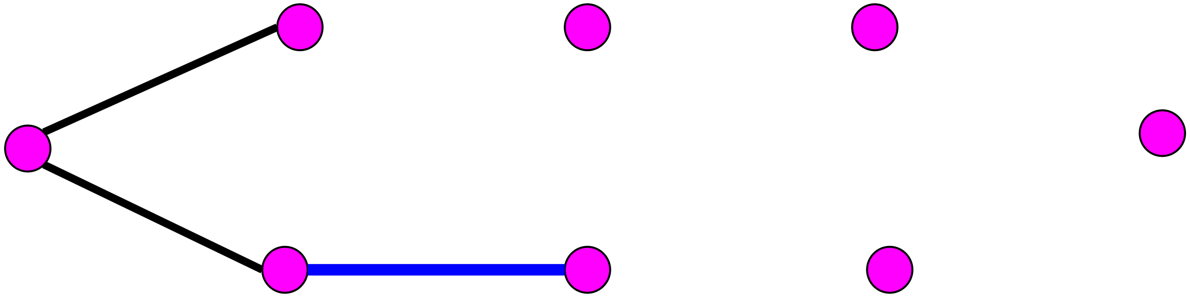


Construction with cost perturbation

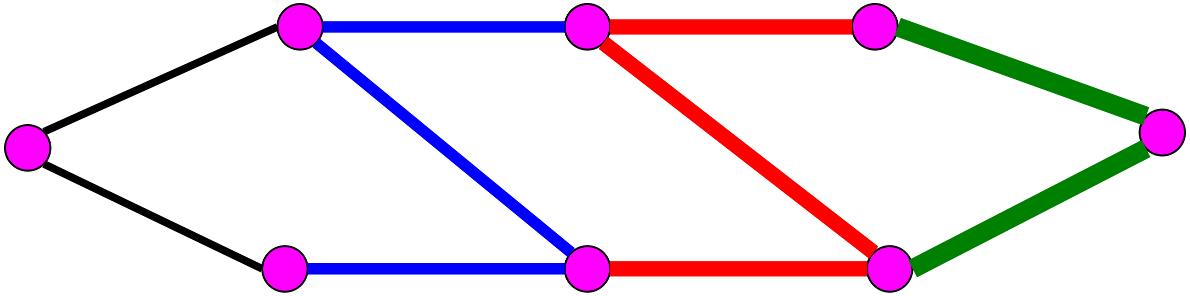


Perturb with costs increasing from bottom to top.

W(⬛) < W(🟦) < W(🟥) < W(🟩)

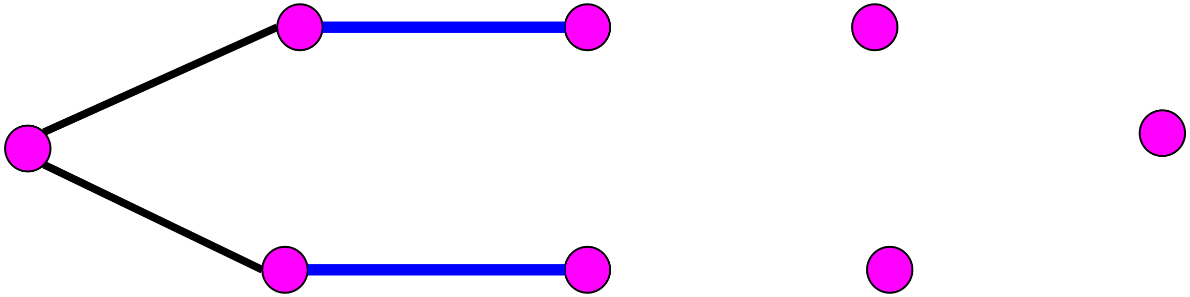


Construction with cost perturbation

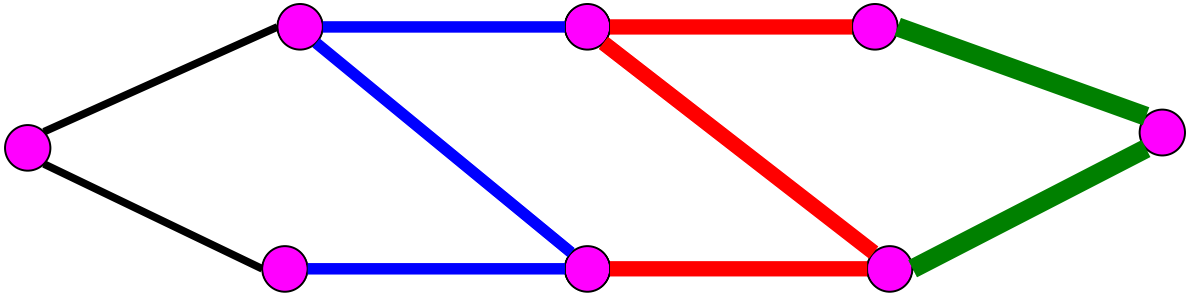


Perturb with costs increasing from bottom to top.

W(⬛) < W(🟦) < W(🟥) < W(🟩)

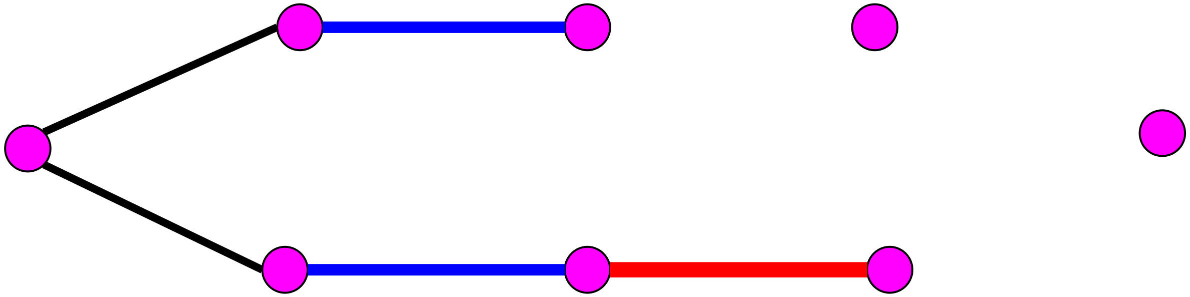


Construction with cost perturbation

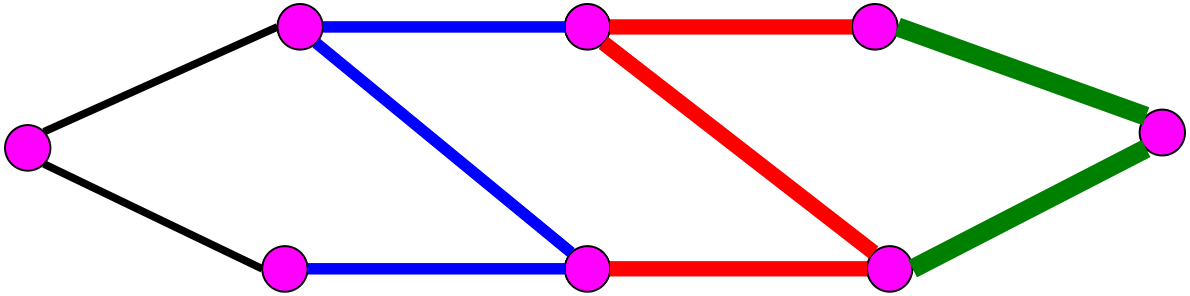


Perturb with costs increasing from bottom to top.

W(▮) < W(🟦) < W(🟥) < W(🟩)

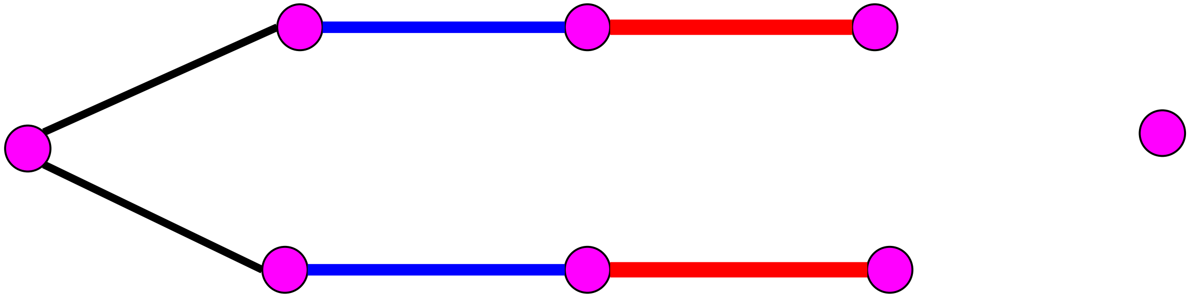


Construction with cost perturbation

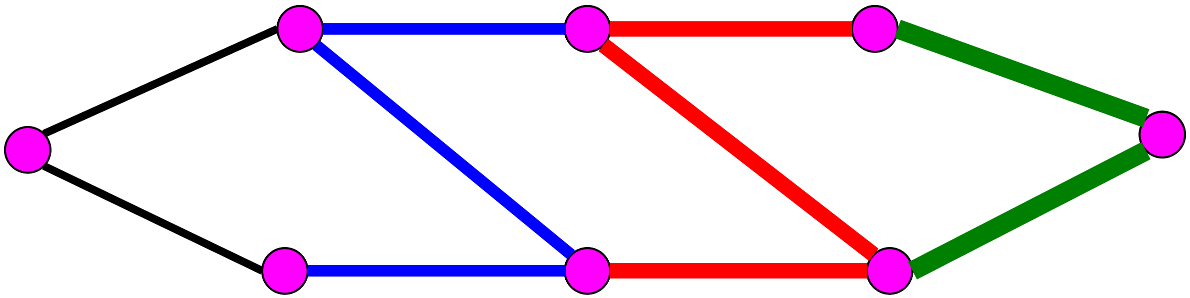


Perturb with costs increasing from bottom to top.

W(▮) < W(🟦) < W(🟥) < W(🟩)

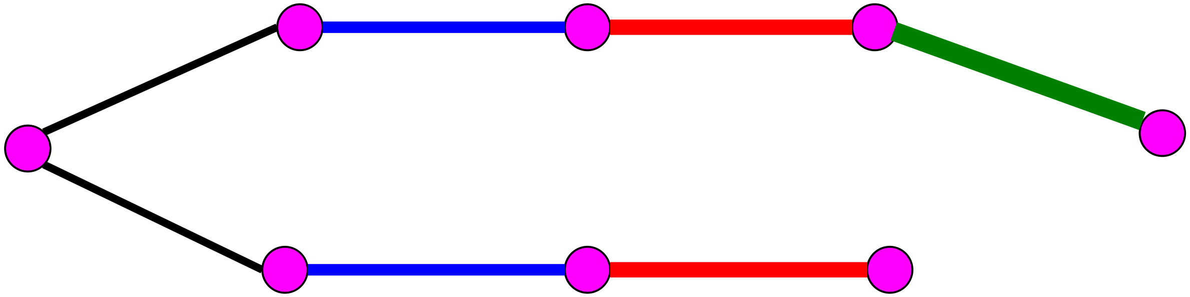


Construction with cost perturbation

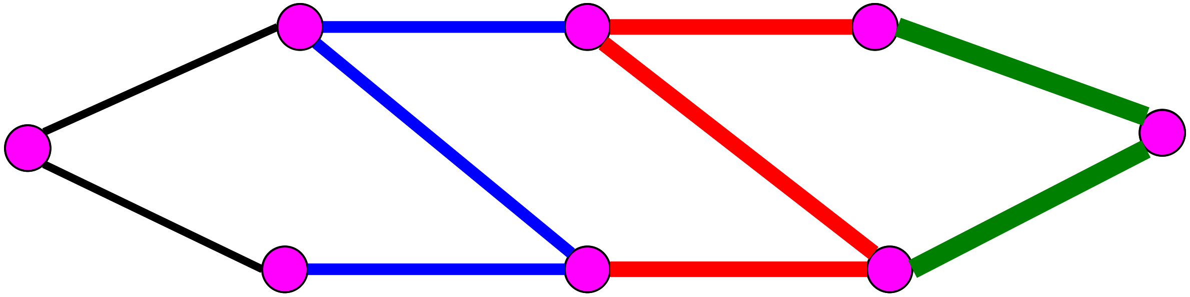


Perturb with costs increasing from bottom to top.

W(⬛) < W(🟦) < W(🟥) < W(🟩)

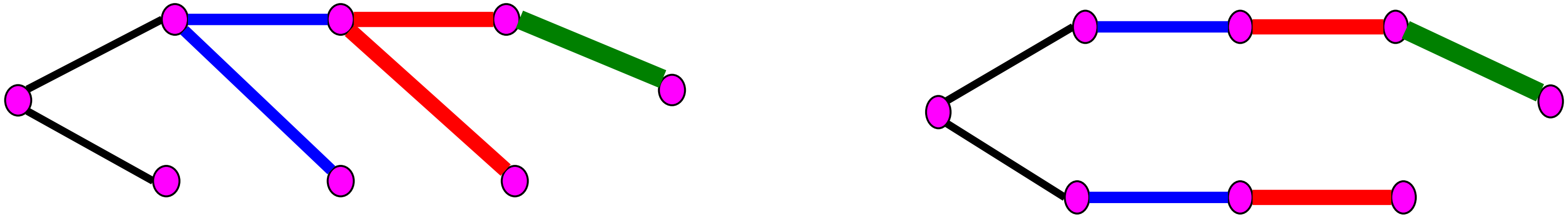


Construction with cost perturbation



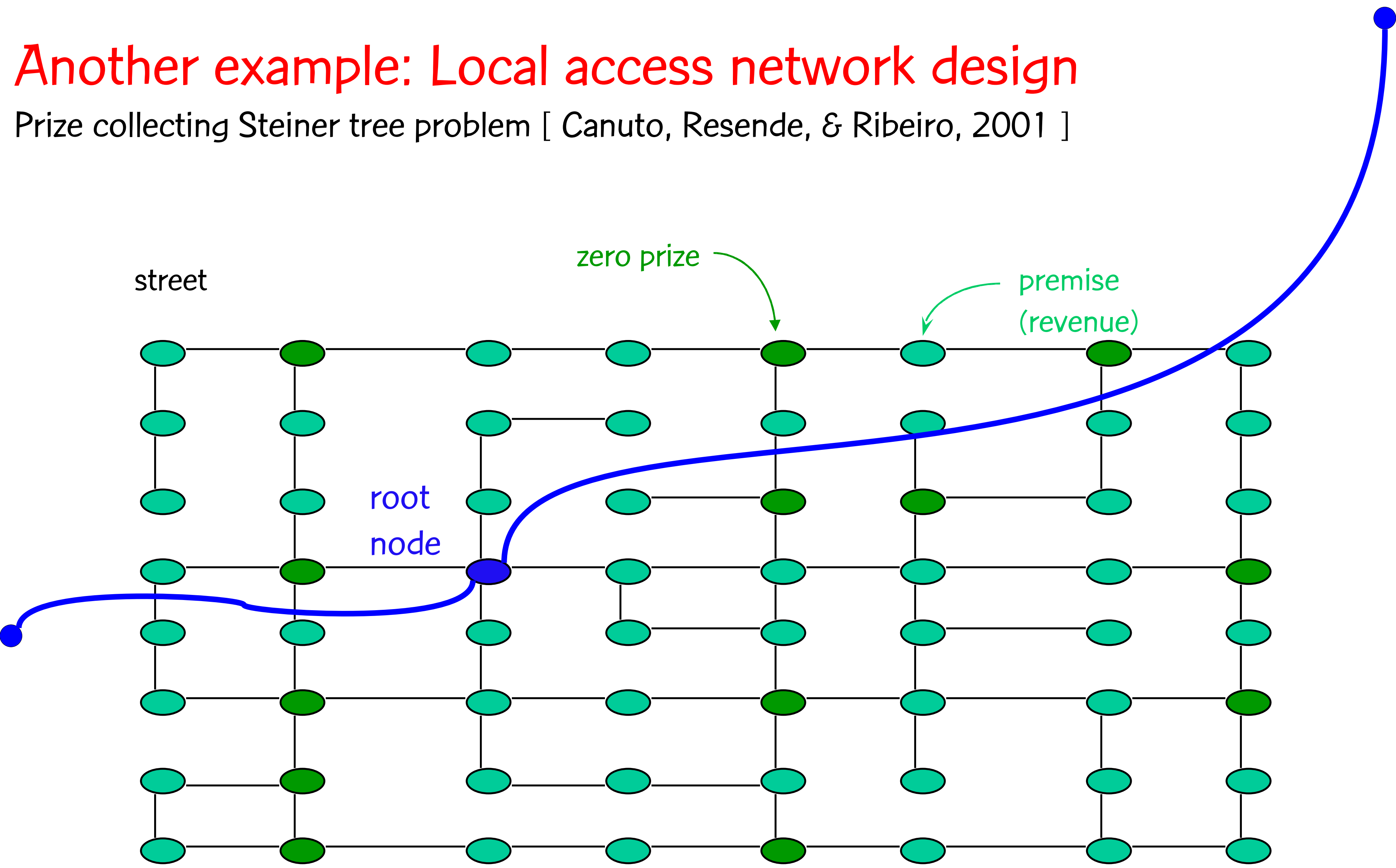
Greedy heuristic generates two different spanning trees.

W(▮) < W(🟦) < W(🟥) < W(🟩)



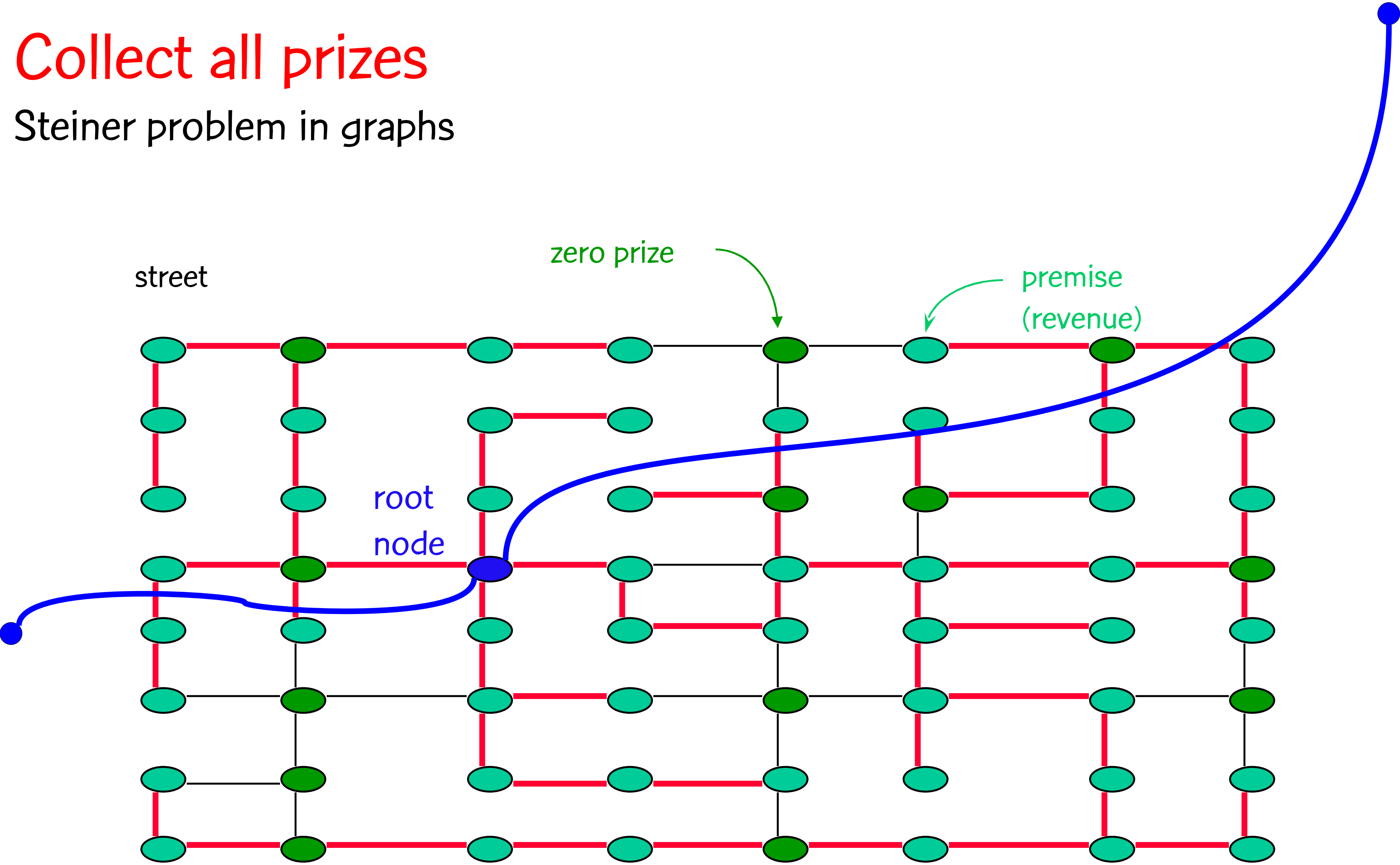
Another example: Local access network design

Prize collecting Steiner tree problem [Canuto, Resende, & Ribeiro, 2001]



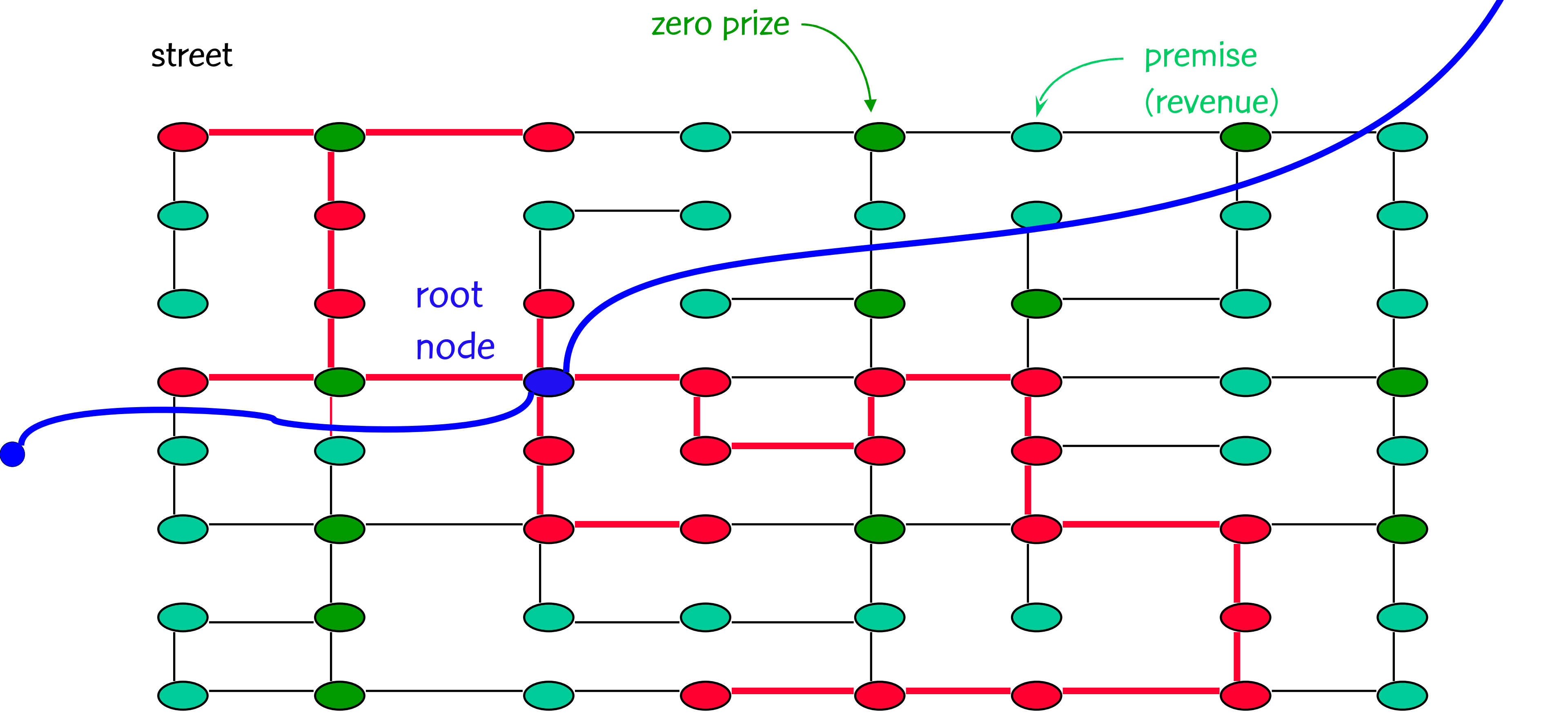
Collect all prizes

Steiner problem in graphs



Collect some prizes

Prize-collecting Steiner Problem in Graphs



Cost perturbation

Prize collecting Steiner tree problem [Canuto, Resende, & Ribeiro, 2001]

- Force 2-approximation algorithm of Goemans & Williamson (GW) to construct different initial solutions for local search:
  - Use original prizes in first iteration and then modified prizes:
- Two strategies for modified prizes:
  - Introduce noise into prizes

```
for i = 1, ..., |V | {
    generate $\beta \in [1 - a, 1 + a]$, for $a > 0$
    $d'(i) = d(i) \times \beta$
}
```

  - Node elimination
    - Set to zero the prizes of α% of the nodes in nodes(GW) $\cap$ nodes(local search)

Reactive GRASP

Prais & Ribeiro (2000)

- When building RCL, what α to use?
  - Fix a some value $0 \leq \alpha \leq 1$
  - Choose α at random (uniformly) at each GRASP iteration.
- Another approach reacts to search ...
  - At each GRASP iteration, a value of the RCL parameter α is chosen from a discrete set of values $[\alpha_1, \alpha_2, ..., \alpha_m]$.
  - The probability that α_k is selected is p_k.
  - Reactive GRASP: adaptively changes the probabilities $[p_1, p_2, ..., p_m]$ to favor values of α that produce good solutions.

Reactive GRASP

Prais & Ribeiro (2000)

- Reactive GRASP for minimization …
- Initially $p_k = 1/m$, for $k = 1,\ldots,m$. (α's are selected uniformly at random)
- Define
  - $F(S^*)$ be the best solution so far
  - A_k be the average value of the solutions obtained with α_k
- Every N_α GRASP iterations, compute
  - $q_k = F(S^*) / A_k$, for $k = 1,\ldots,m$
  - $p_k = q_k / \text{sum}(q_i \mid i = 1,\ldots,m)$

Reactive GRASP

Prais & Ribeiro (2000)

- Reactive GRASP for minimization …
- Initially $p_k = 1/m$, for $k = 1,\ldots,m$. (α's select uniformly at random)
- Define
  - F(S*) be the best solution so far
  - A_k be the average value of the solutions obtained with α_k
- Every N_α GRASP iterations, compute
  - $q_k = F(S^*) / A_k$, for $k = 1,\ldots,m$
  - $p_k = q_k / \text{sum}(q_i \mid i = 1,\ldots,m)$

The more suitable is α_k, the larger is q_k, and consequently p_k, making α_k more likely to chosen.

Adaptive memory construction

[Fleurent & Glover,1999]

- Propose an RCL-based construction that makes use of memory structures.
- An elite set S of diverse high-quality solutions found during the search is maintained. To be in S a solution must be:
  - better than the best solution in S, or
  - better than worst and sufficiently different from all elite solutions
- This elite set is used during construction.

Adaptive memory construction

[Fleurent & Glover,1999]

- A strongly determined variable is one that cannot be changed without eroding the objective or changing significantly the other variables.
- A consistent variable is one that receives a particular value in a large portion of the elite solution set.
- Let intensity I(e) be a measure of the strongly determined and consistent features of candidate e, i.e. I(e) grows as e resembles solutions in S.

Adaptive memory construction

[Fleurent & Glover,1999]

- Use intensity function during construction:
  - – Recall $g(e)$ is greedy function
  - – Let $E(e) = F(\ g(e), I(e)\ )$, e.g. $E(e) = \lambda\ g(e) + I(e)$
  - – Bias selection from RCL to those elements e with a high E(e), i.e.
    - prob(selecting e) = E(e) / sum(E(s) | $s \in S$)
- Initially λ should be kept large since memory structure is young. It should be slowly reduced until $\lambda = 0$, where it should be kept.

Local search in GRASP

Local search within GRASP

- First improving vs. best improving:
  - First improving is usually faster.
  - Premature convergence to low quality local optimum is more likely to occur with best improving.
- Hashing to avoid cycling or repeated application of local search to same solution built in the construction phase: Woodruff & Zemel (1993), Ribeiro et. al (1997) (query optimization), Martins et al. (2000) (Steiner problem in graphs)

Local search within GRASP

- Filtering to avoid application of local search to low quality solutions, only promising unvisited solutions are investigated: Feo, Resende, & Smith (1994), Prais & Ribeiro (2000) (traffic assignment), Martins et. al (2000) (Steiner problem in graphs)
- Extended quick-tabu local search to overcome premature convergence: Souza, Duhamel, & Ribeiro (2003) (capacitated minimum spanning tree, better solutions for largest benchmark problems)

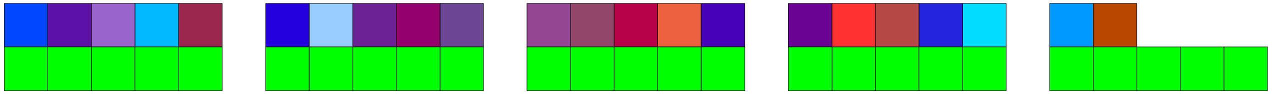
Local search within GRASP

- Variable Neighborhood Descent (VND) to speedup search and to overcome optimality w.r.t. to simple (first) neighborhood: Ribeiro, Uchoa, & Werneck (2002) (Steiner problem in graphs)

GRASP VND local search

example: scheduling of multi-grouped units

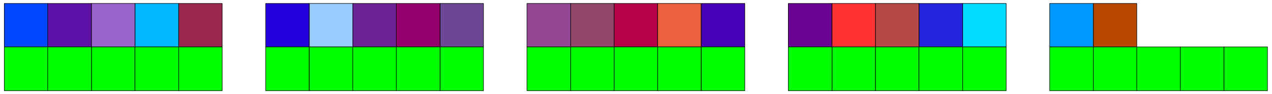
- Input: Assignment of units to periods:



GRASP VND local search

example: scheduling of multi-grouped units

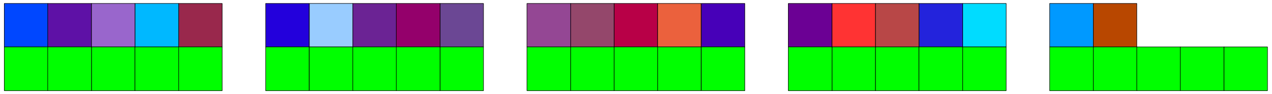
- Local search: Examine neighborhood of current solution. If better solution found, make it current solution.



GRASP VND local search

example: scheduling of multi-grouped units

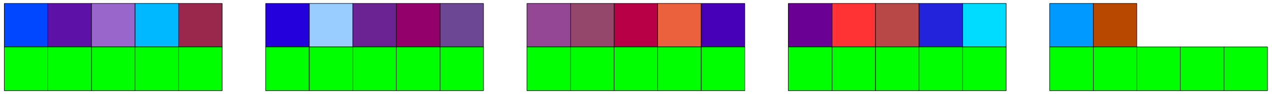
- Three neighborhoods: Swap units, move unit, swap periods.



GRASP VND local search

example: scheduling of multi-grouped units

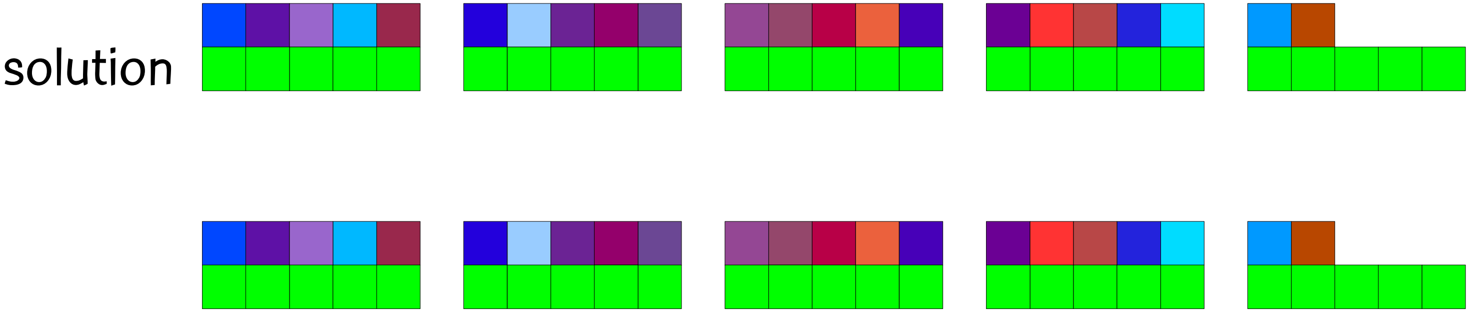
- Swap units neighborhood: Swaps places of two units assigned to distinct periods.



GRASP VND local search

example: scheduling of multi-grouped units

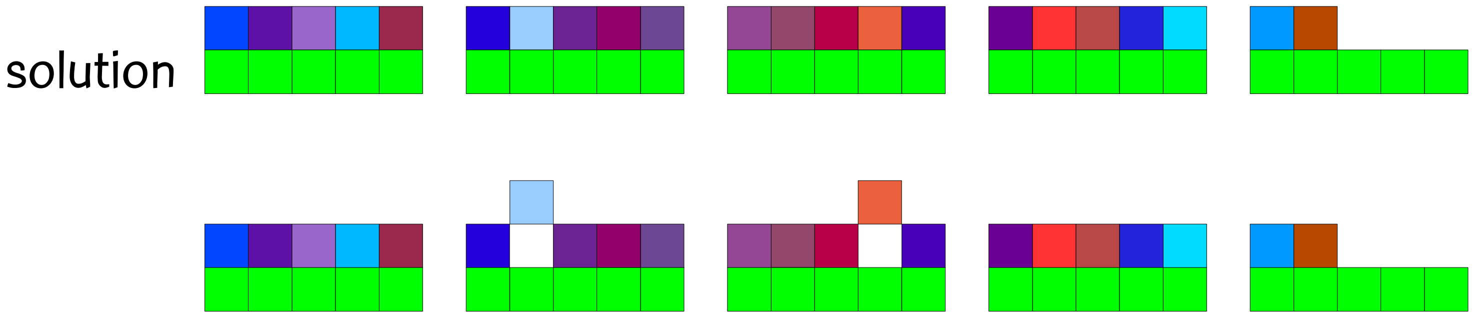
- Swap units neighborhood: Swaps places of two units assigned to distinct periods.



GRASP VND local search

example: scheduling of multi-grouped units

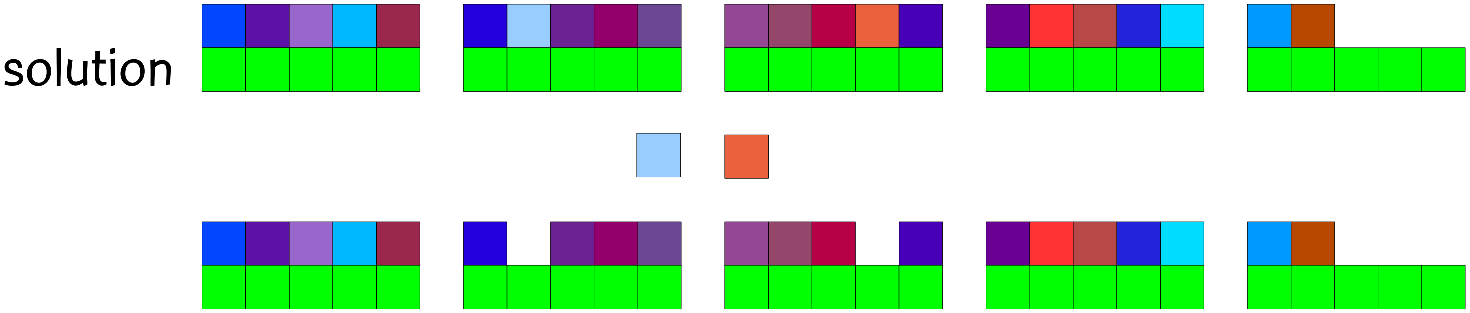
- Swap units neighborhood: Swaps places of two units assigned to distinct periods.



GRASP VND local search

example: scheduling of multi-grouped units

- Swap units neighborhood: Swaps places of two units assigned to distinct periods.

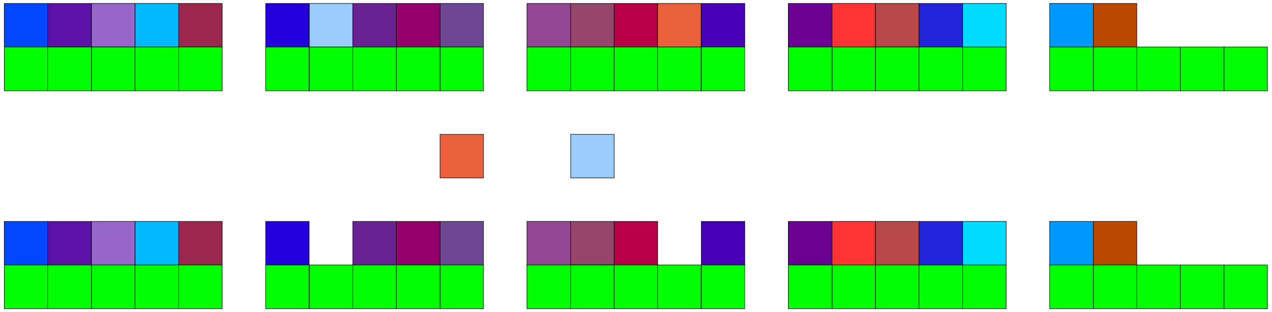


GRASP VND local search

example: scheduling of multi-grouped units

- Swap units neighborhood: Swaps places of two units assigned to distinct periods.

solution



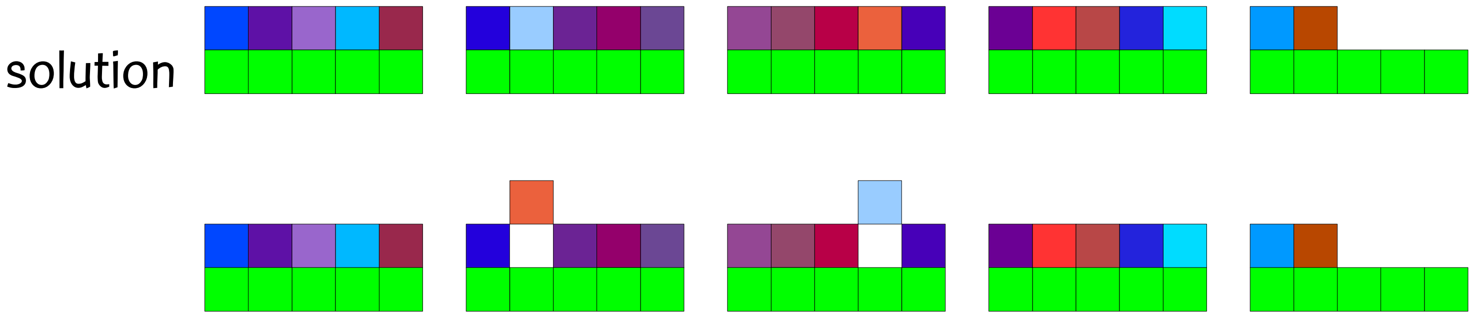
Aug. 2007

Short course on GRASP

GRASP VND local search

example: scheduling of multi-grouped units

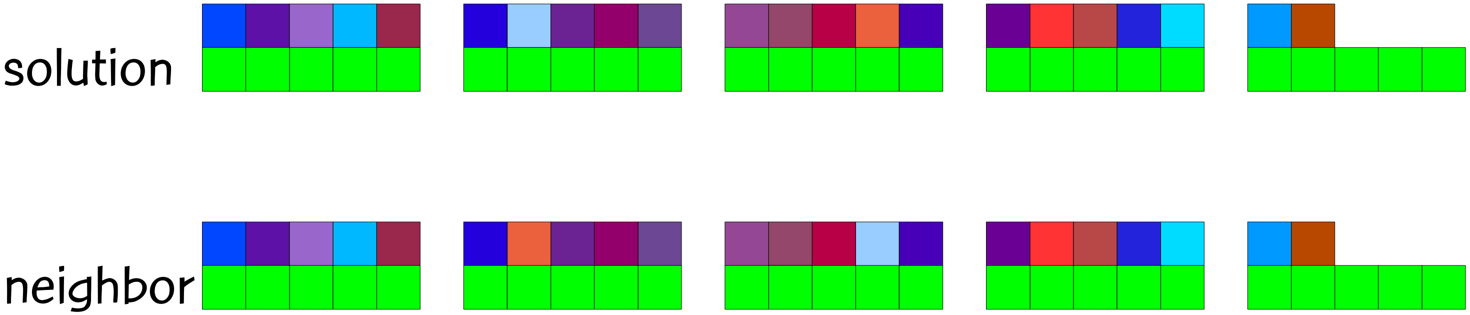
- Swap units neighborhood: Swaps places of two units assigned to distinct periods.



GRASP VND local search

example: scheduling of multi-grouped units

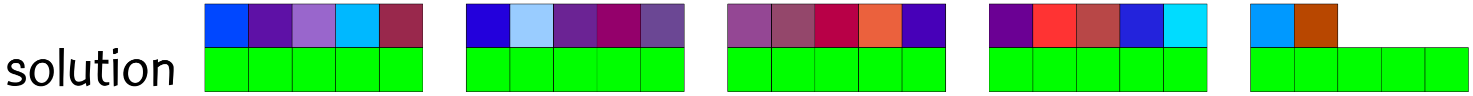
- Swap units neighborhood: Swaps places of two units assigned to distinct periods.



GRASP VND local search

example: scheduling of multi-grouped units

- Move unit neighborhood: Moves unit from current period to available period.

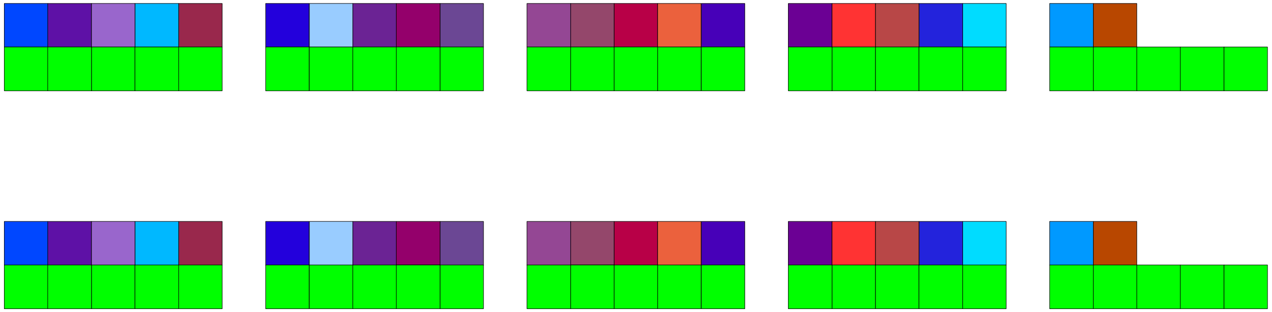


GRASP VND local search

example: scheduling of multi-grouped units

- Move unit neighborhood: Moves unit from current period to available period.

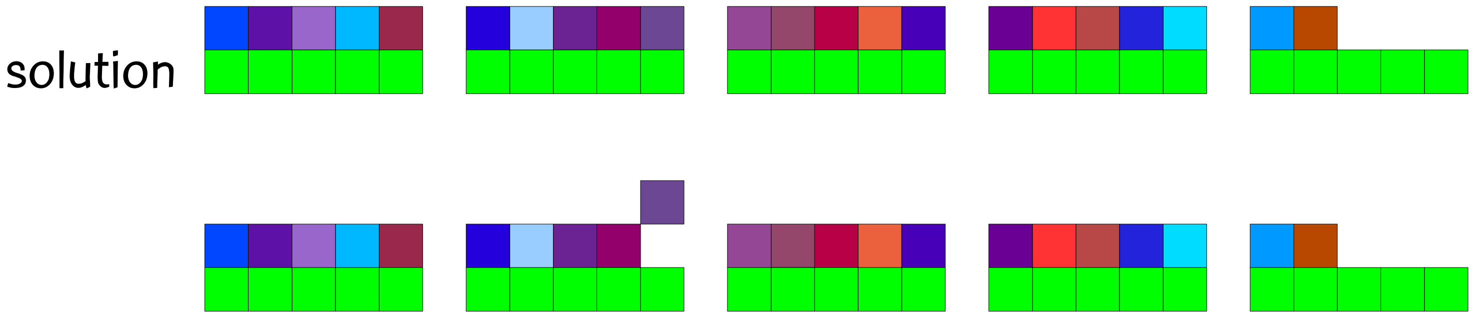
solution



GRASP VND local search

example: scheduling of multi-grouped units

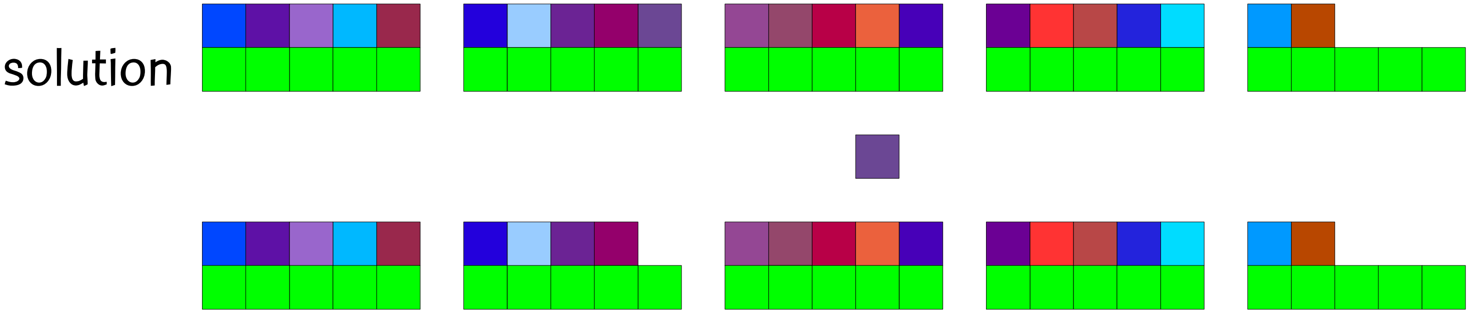
- Move unit neighborhood: Moves unit from current period to available period.



GRASP VND local search

example: scheduling of multi-grouped units

- Move unit neighborhood: Moves unit from current period to available period.

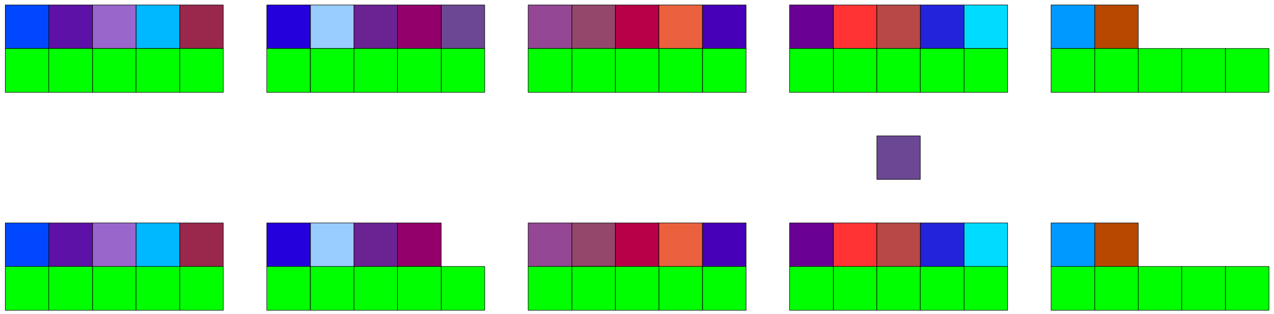


GRASP VND local search

example: scheduling of multi-grouped units

- Move unit neighborhood: Moves unit from current period to available period.

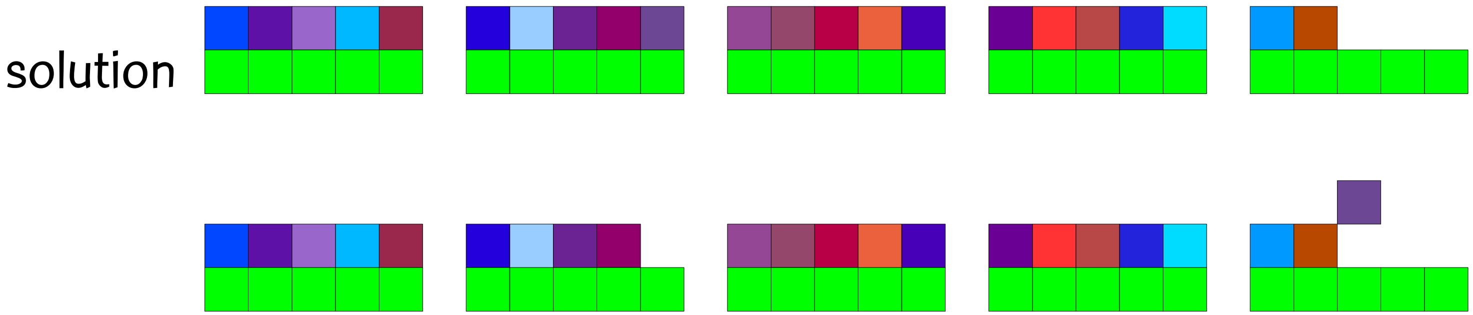
solution



GRASP VND local search

example: scheduling of multi-grouped units

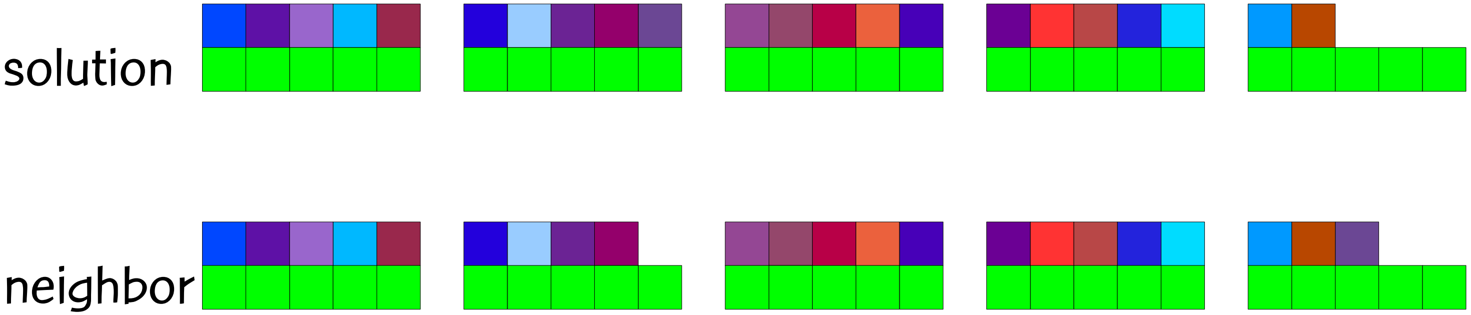
- Move unit neighborhood: Moves unit from current period to available period.



GRASP VND local search

example: scheduling of multi-grouped units

- Move unit neighborhood: Moves unit from current period to available period.

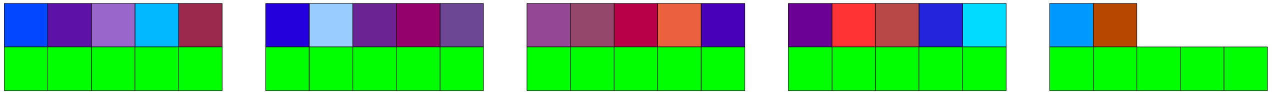


GRASP VND local search

example: scheduling of multi-grouped units

- Swap periods neighborhood: Swap all units in period i with all units in period j.

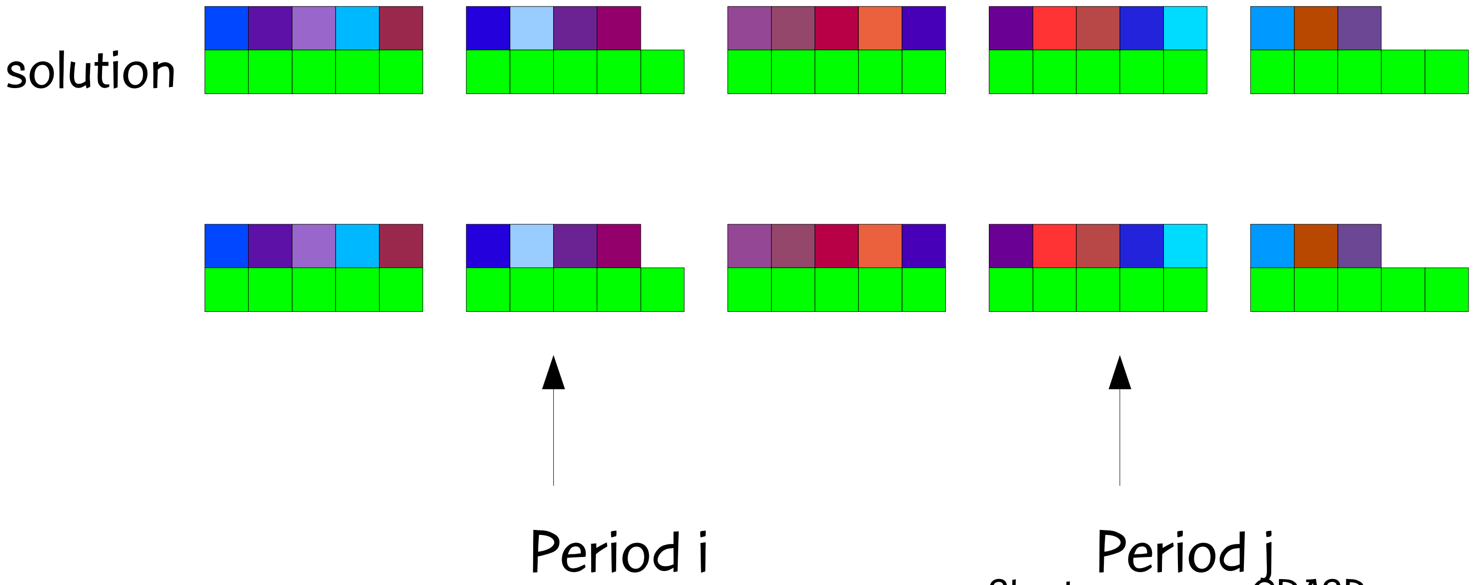
solution



GRASP VND local search

example: scheduling of multi-grouped units

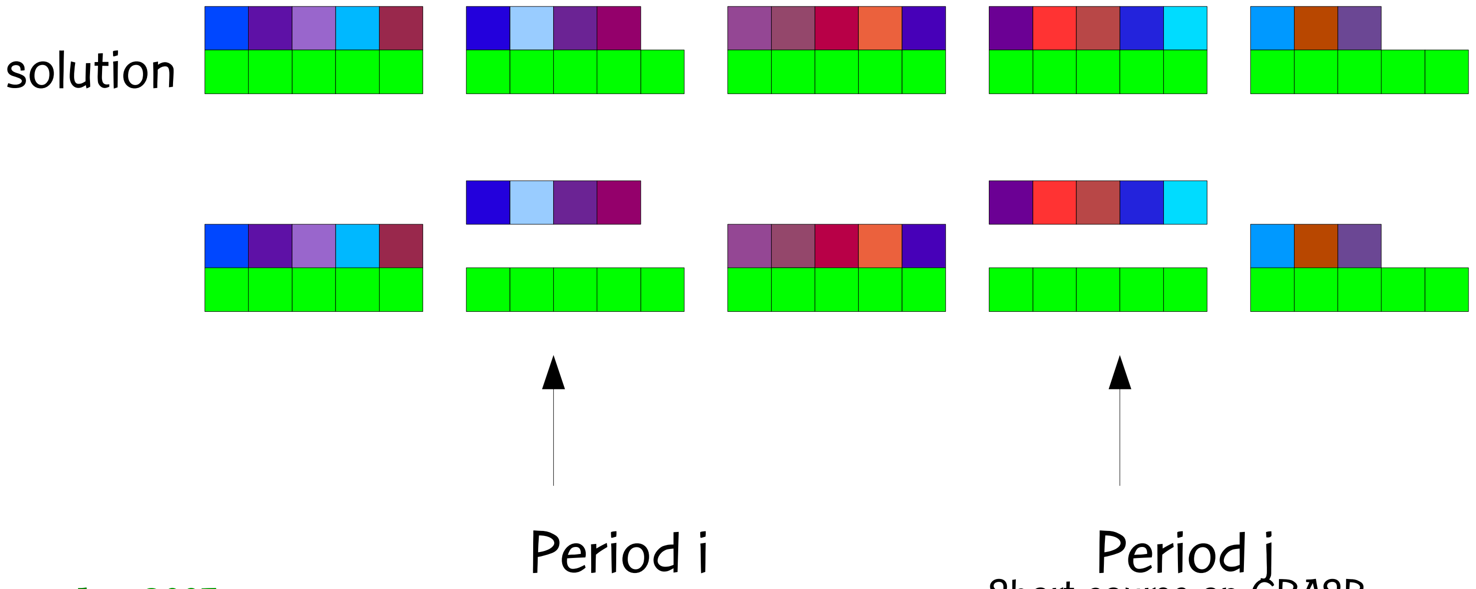
- Swap periods neighborhood: Swap all units in period i with all units in period j.



GRASP VND local search

example: scheduling of multi-grouped units

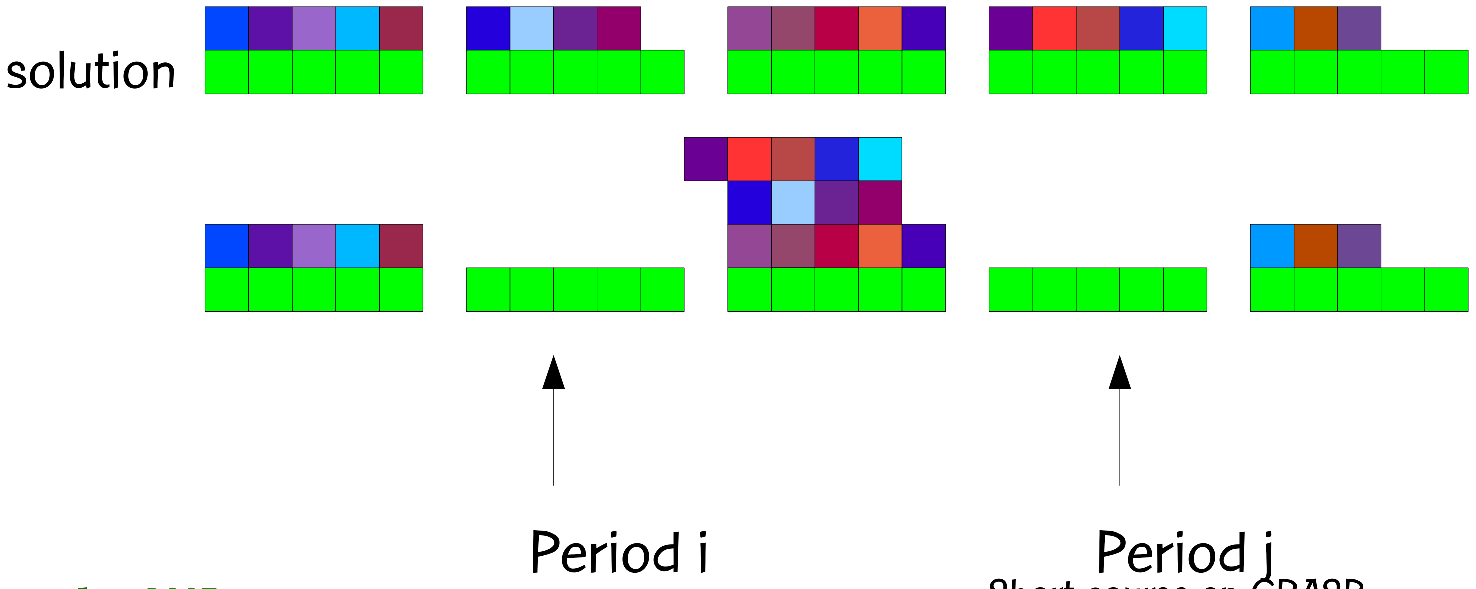
- Swap periods neighborhood: Swap all units in period i with all units in period j.



GRASP VND local search

example: scheduling of multi-grouped units

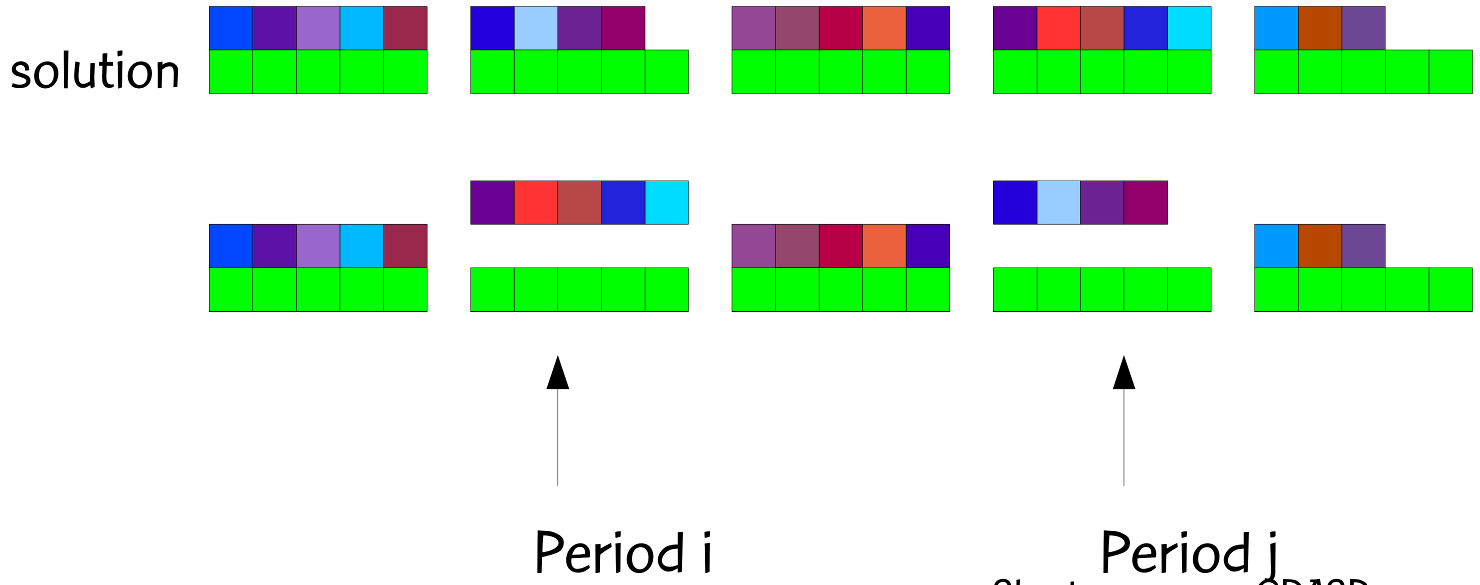
- Swap periods neighborhood: Swap all units in period i with all units in period j.



GRASP VND local search

example: scheduling of multi-grouped units

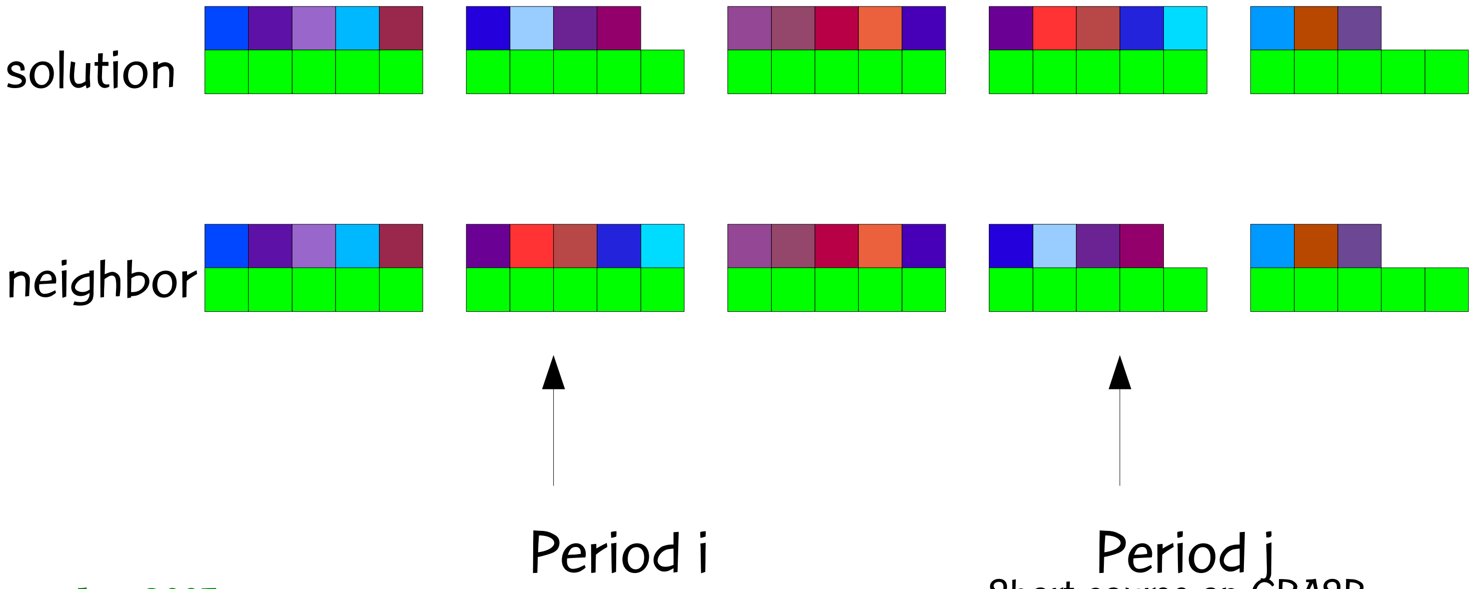
- Swap periods neighborhood: Swap all units in period i with all units in period j.



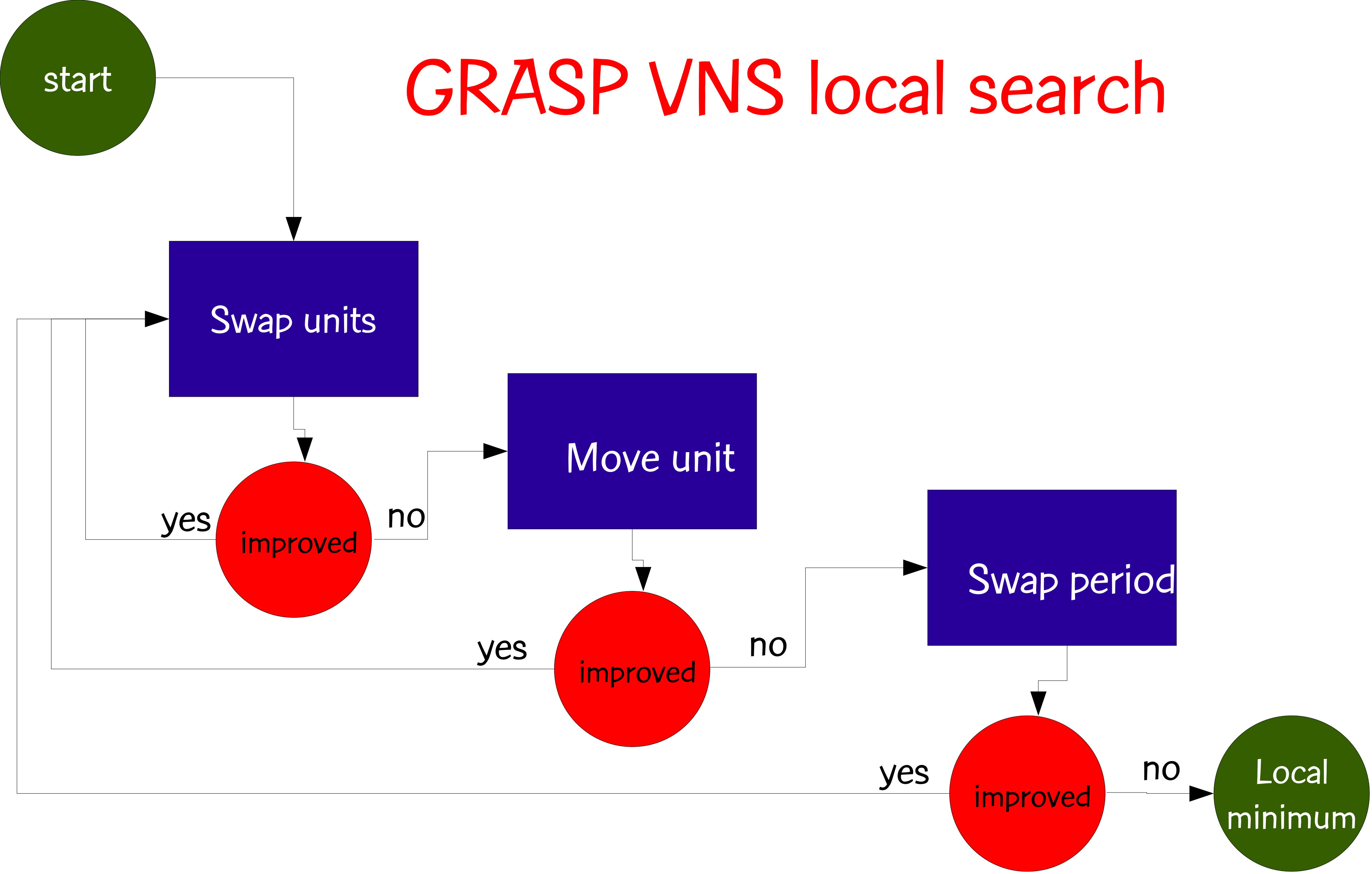
GRASP VND local search

example: scheduling of multi-grouped units

- Swap periods neighborhood: Swap all units in period i with all units in period j.



GRASP VNS local search



Day 2 of Short Course on GRASP

Summary

- Day 1
  - Combinatorial opt. & metaheuristics
  - Local search
  - Greedy algorithm
  - Basic GRASP
    - Construction
    - Local search within GRASP
  - Some extensions
    - Reactive GRASP
    - Memory in construction
- Day 2
  - Prob. distribution of running time
  - Time-to-target plots
  - Path-relinking (PR) & Evolutionary PR (EvPR)
  - GRASP with PR
  - GRASP with EvPR
  - Parallel GRASP
    - Independent threads
    - Cooperative threads
  - Implementation & testing

Probability distribution of GRASP solution time

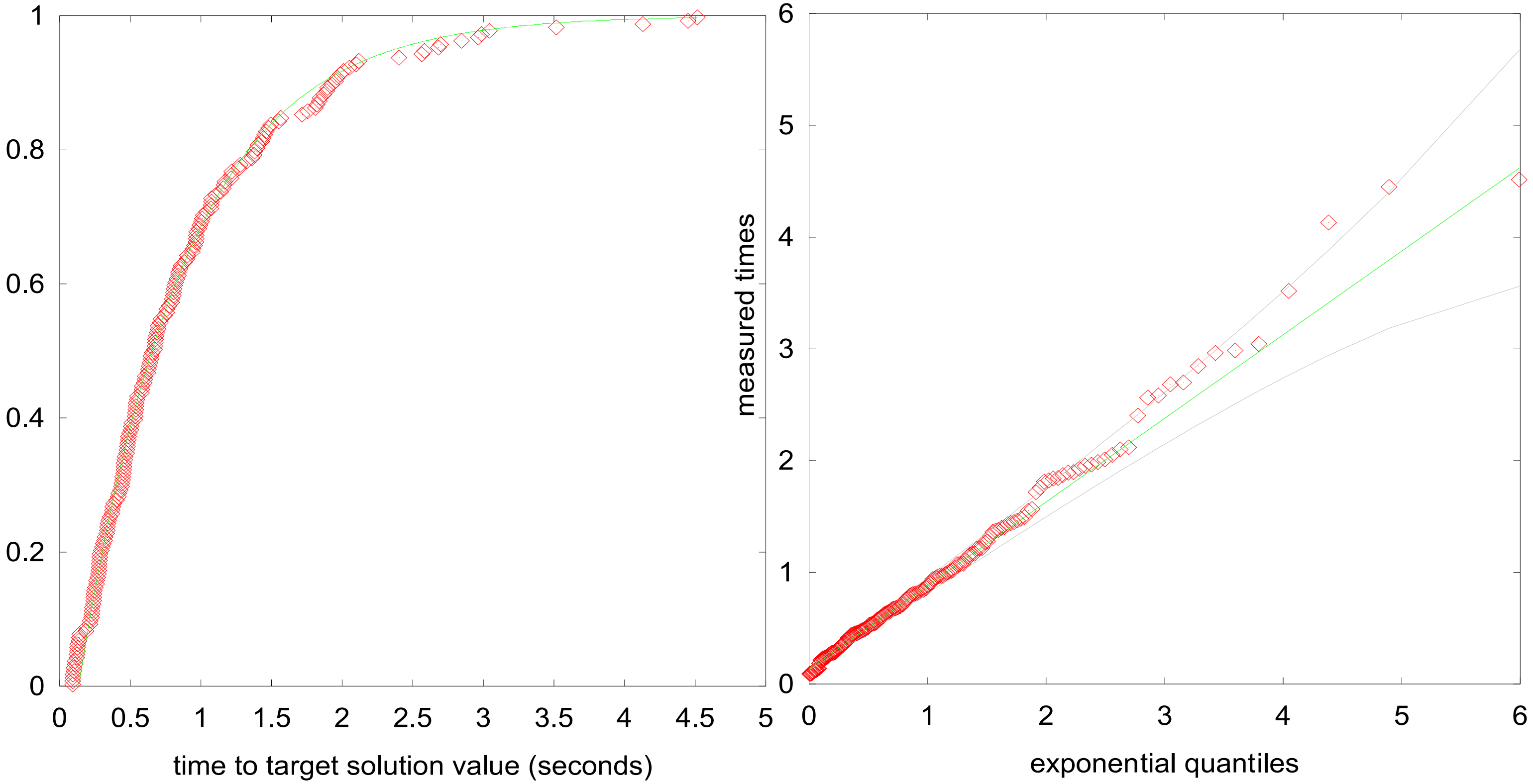
Distribution of running time

- Probability distribution of time-to-target-solution-value: experimental plots
- Select an instance and a target value.
- For each variant of heuristic:
  - Perform 200 runs using different seeds.
  - Stop when a solution value at least as good as the target is found.
  - For each run, measure the time-to-target-value.
  - Plot the probabilities of finding a solution at least as good as the target value within some computation time.

Distribution of running time

- Probability distribution of time-to-target-solution-value: Aiex, Resende, & Ribeiro (2002) have shown experimentally that GRASP running time fits a shifted exponential distribution.

Distribution of running time



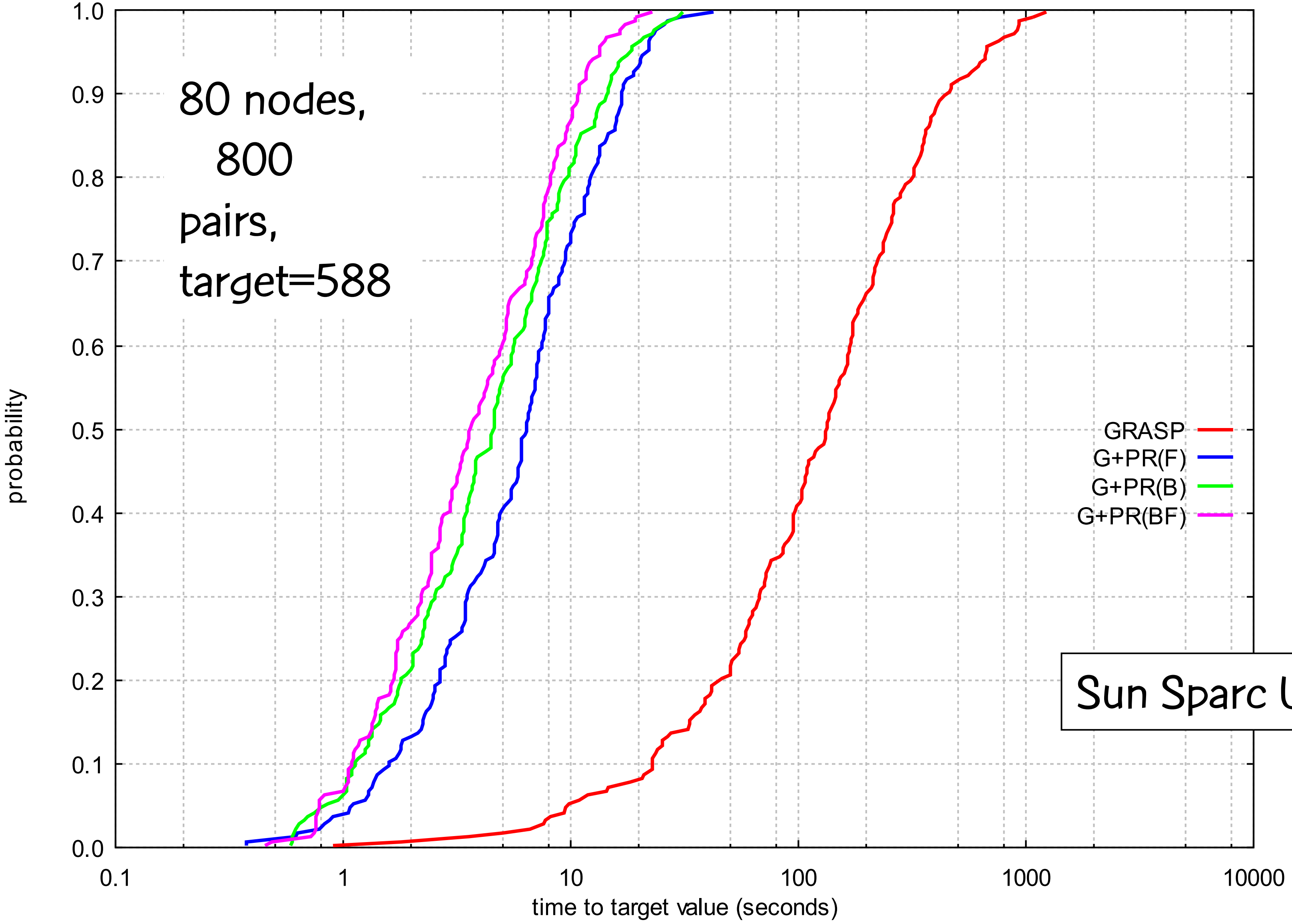
Random variable time-to-target-solution value fits a two-parameter exponential distribution.

Time to target (TTT) plots

TTT plots

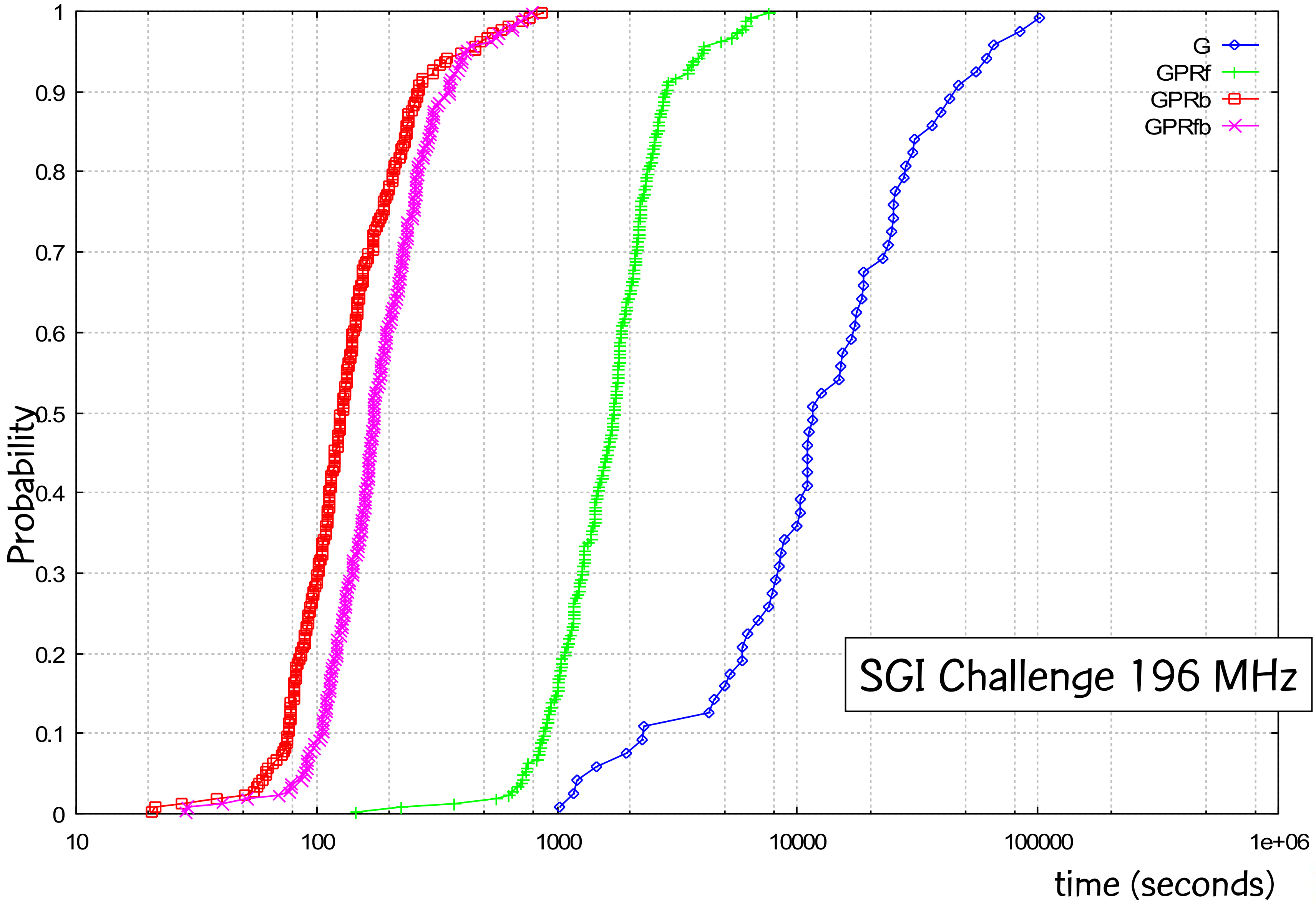
- Time to target (TTT) plots are useful for understanding the behavior of a heuristic's running time.
- Can be used to compare
  - Different variants of a heuristic
  - Different heuristics
  - Same heuristic on problems with varying degrees of difficulty.
  - Parallel heuristics with different number of processors.

Comparing different heuristics



Sun Sparc Ultra 1

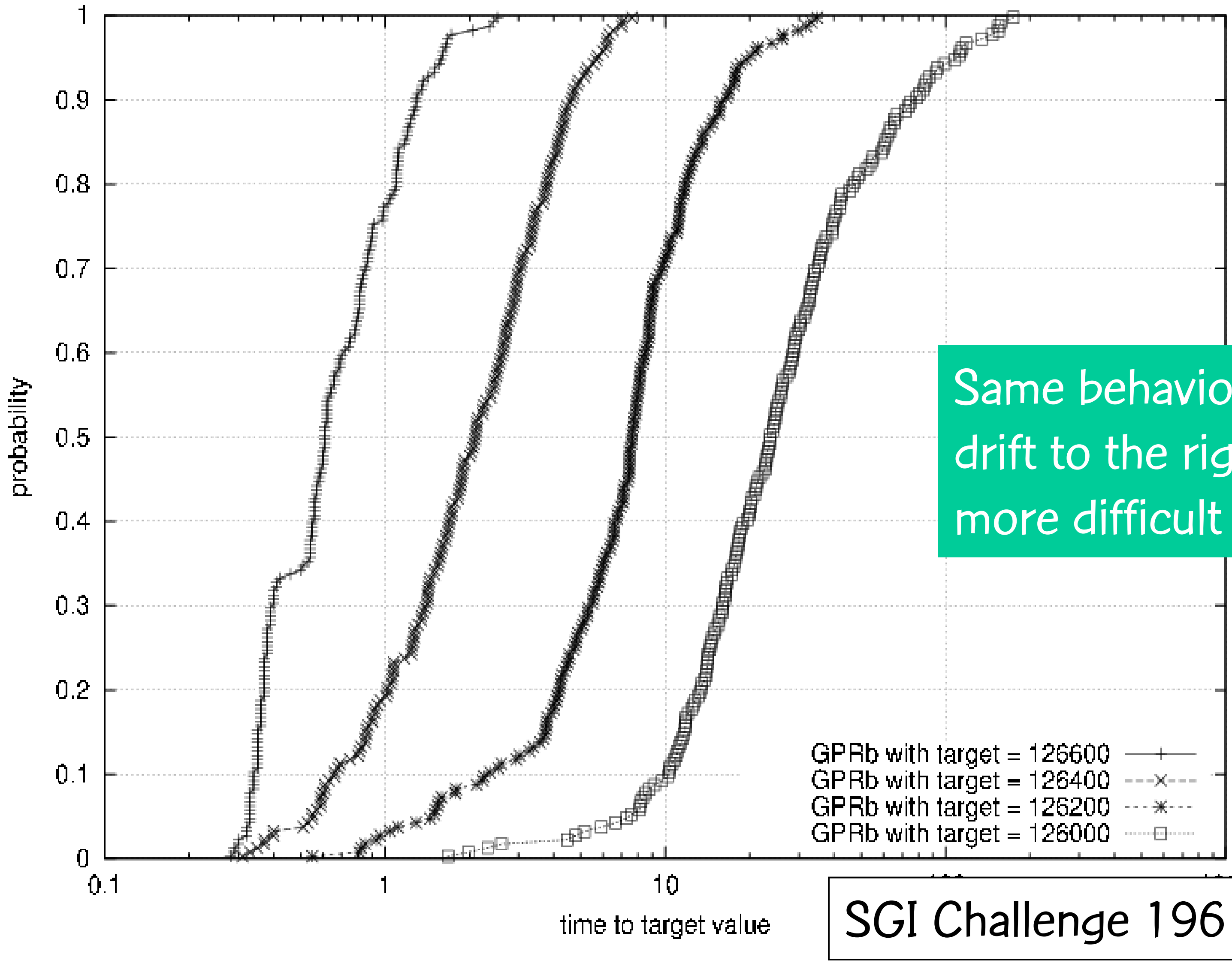
Comparing different heuristics



Aug. 2007

Short course on GRASP

Comparing same heuristic on harder and harder target values



Same behavior, plots drift to the right for more difficult targets

SGI Challenge 196 MHz

Perl program to produce TTT plots

- Aiex, Resende, and Ribeiro (2007) describe a perl program to produce TTT plots:
  - Superimposed theoretical and empirical cumulative probability distributions
  - Q-Q plot superimposed with variability information
- http://www.research.att.com/~mgcr/tttplots

Perl program to produce TTT plots

- To run: `perl tttplots.pl -f input_filename`
- Where **`input_filename.dat`** is the input file with N CPU time data points, one per line.
- The program produces the distribution data files, gnuplot files to produce the plots, as well as PostScript files of the plots.

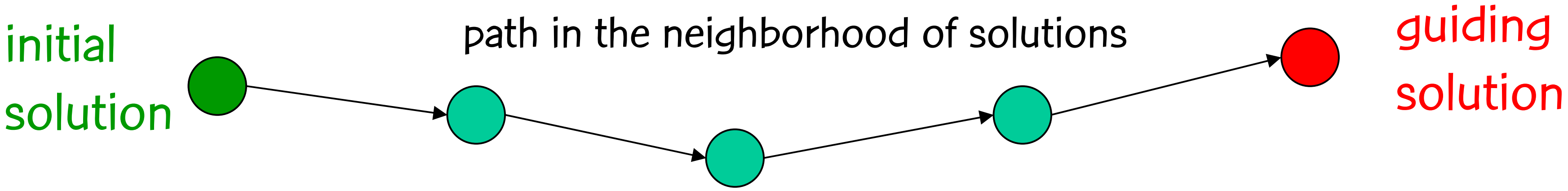
Path-relinking (PR)

Path-relinking

- Intensification strategy exploring trajectories connecting elite solutions (Glover, 1996)
- Originally proposed in the context of tabu search and scatter search.
- Paths in the solution space leading to other elite solutions are explored in the search for better solutions.

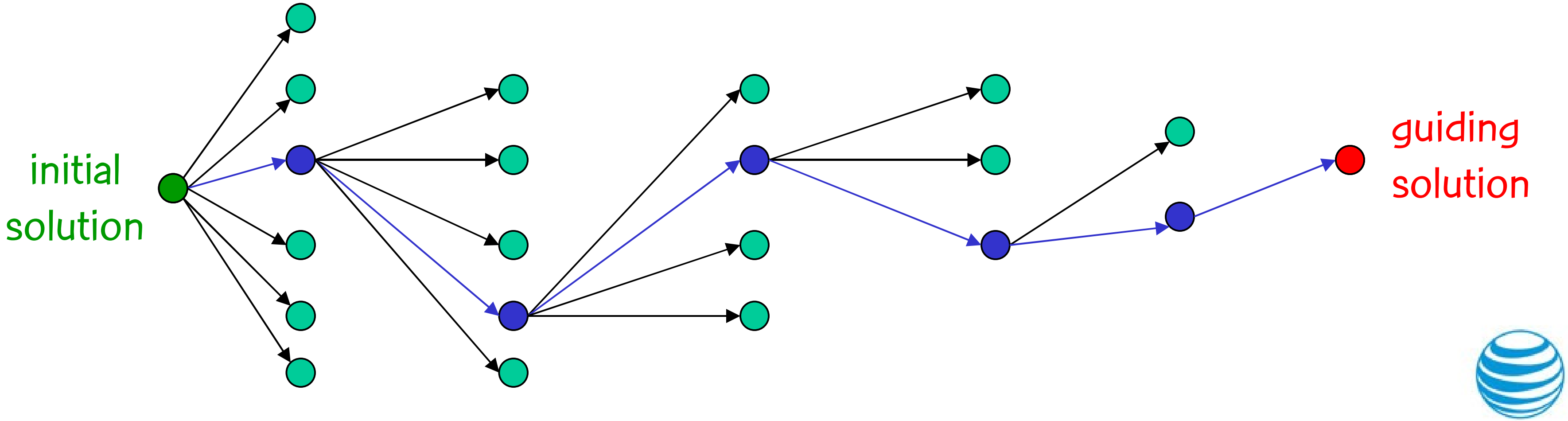
Path-relinking

- Exploration of trajectories that connect high quality (elite) 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 evaluated and the best move is selected:

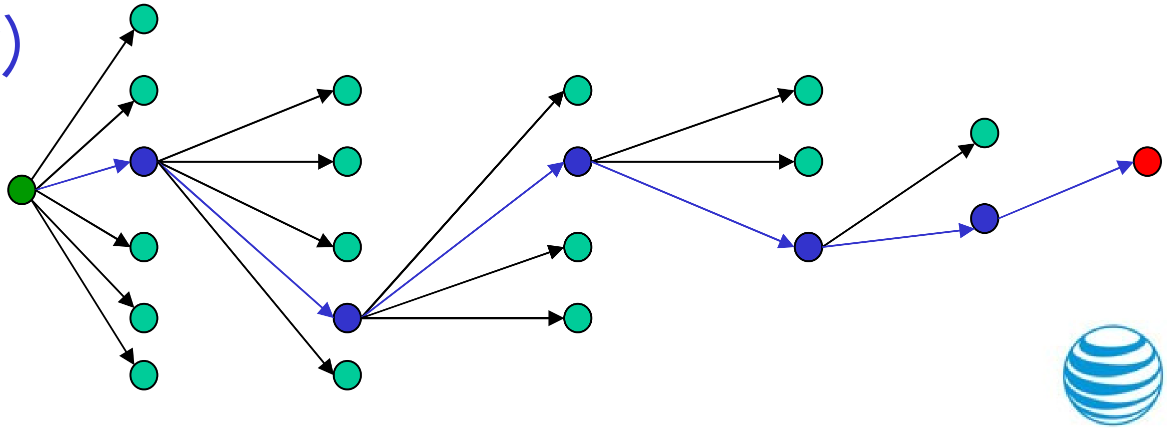


Path-relinking

Solutions x and y to be combined.

$\Delta(x,y)$: symmetric difference between x and y

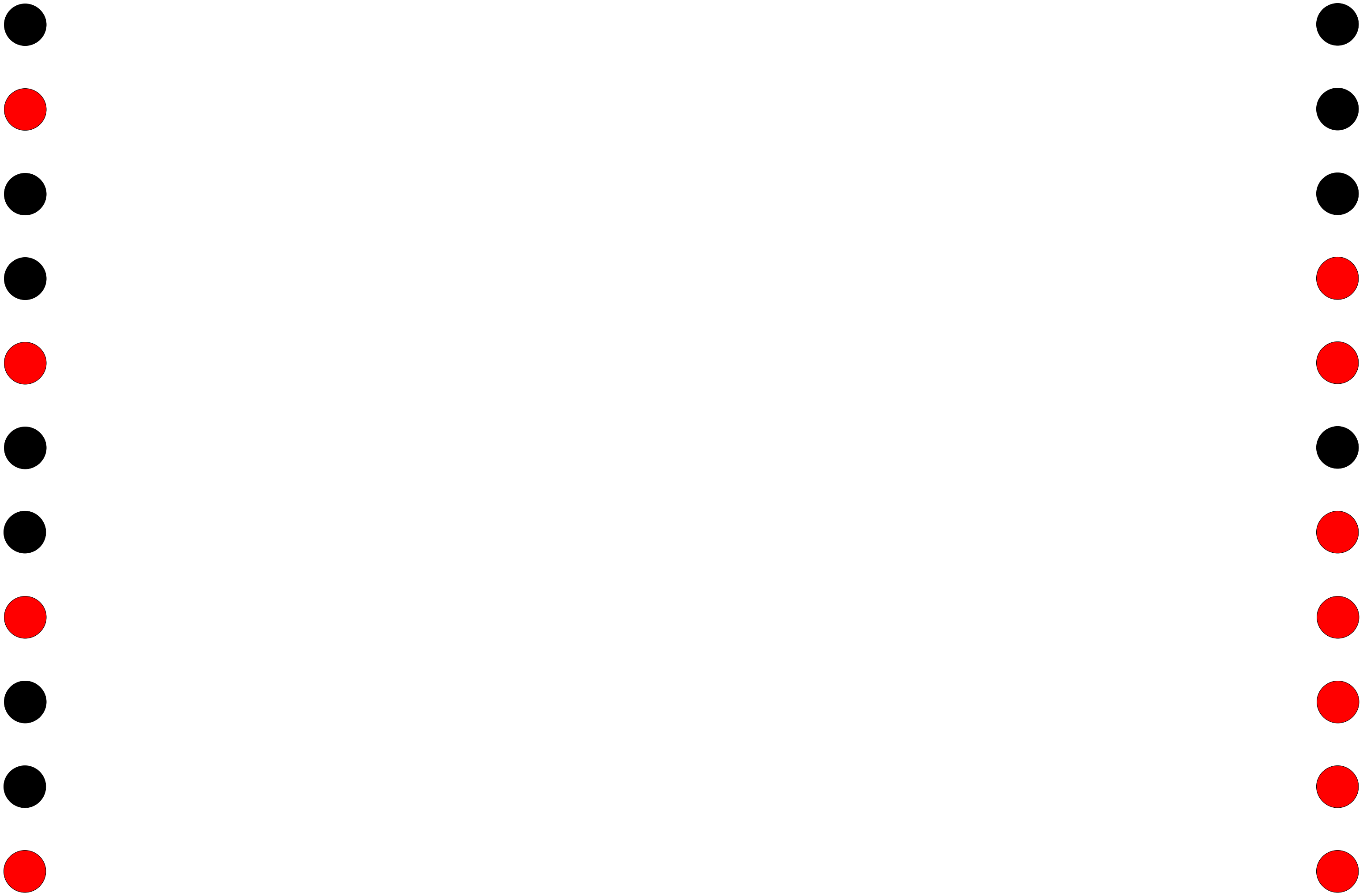
```
while ( $|\Delta(x,y)| > 0$ ) {
    1: evaluate corresponding moves in $\Delta(x,y)$
    2: make best move
    3: update $\Delta(x,y)$
}
```



PR example

starting solution

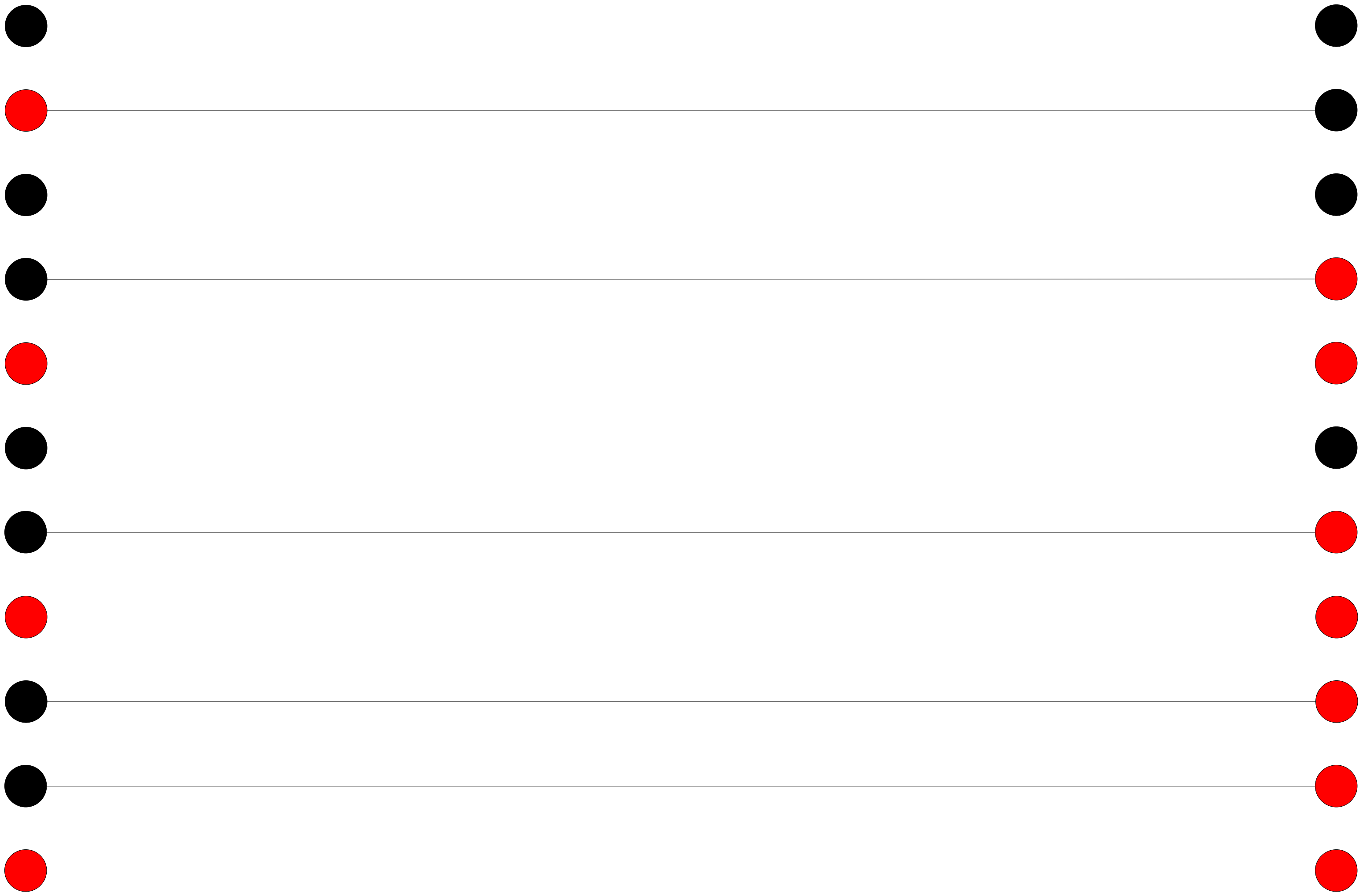
guiding solution



starting solution x

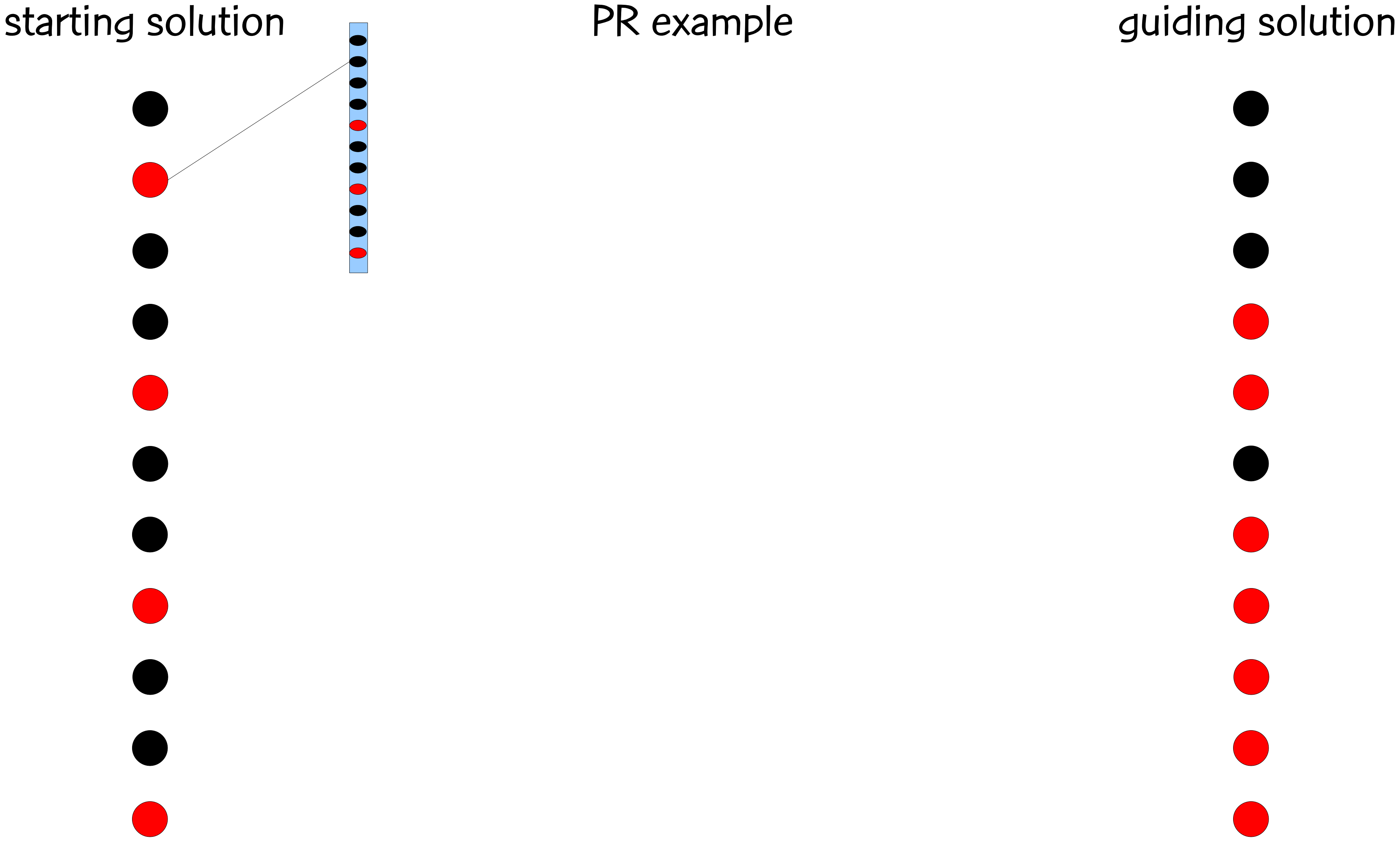
PR example

guiding solution y

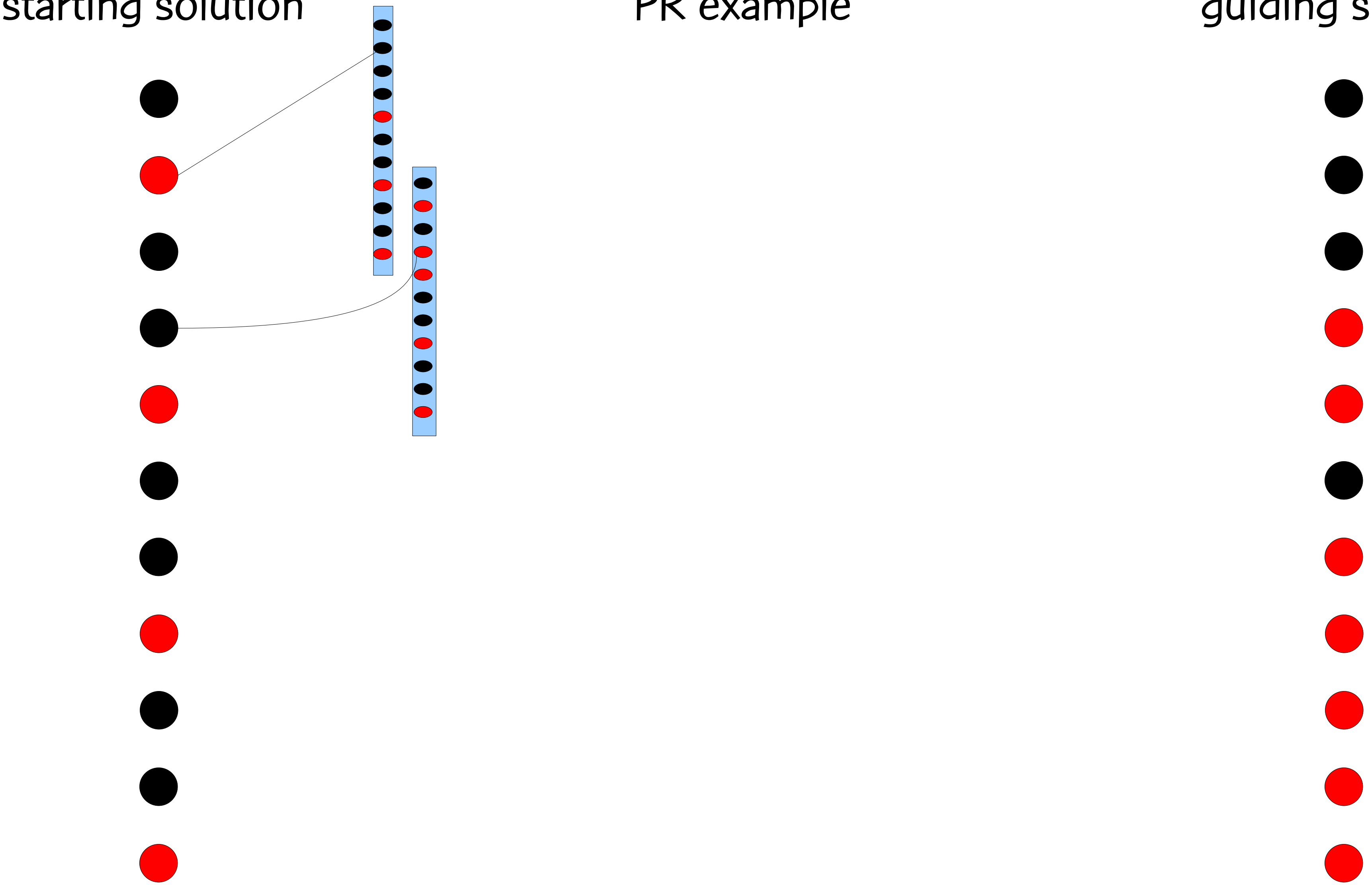


$|\Delta(x,y)| = 5$

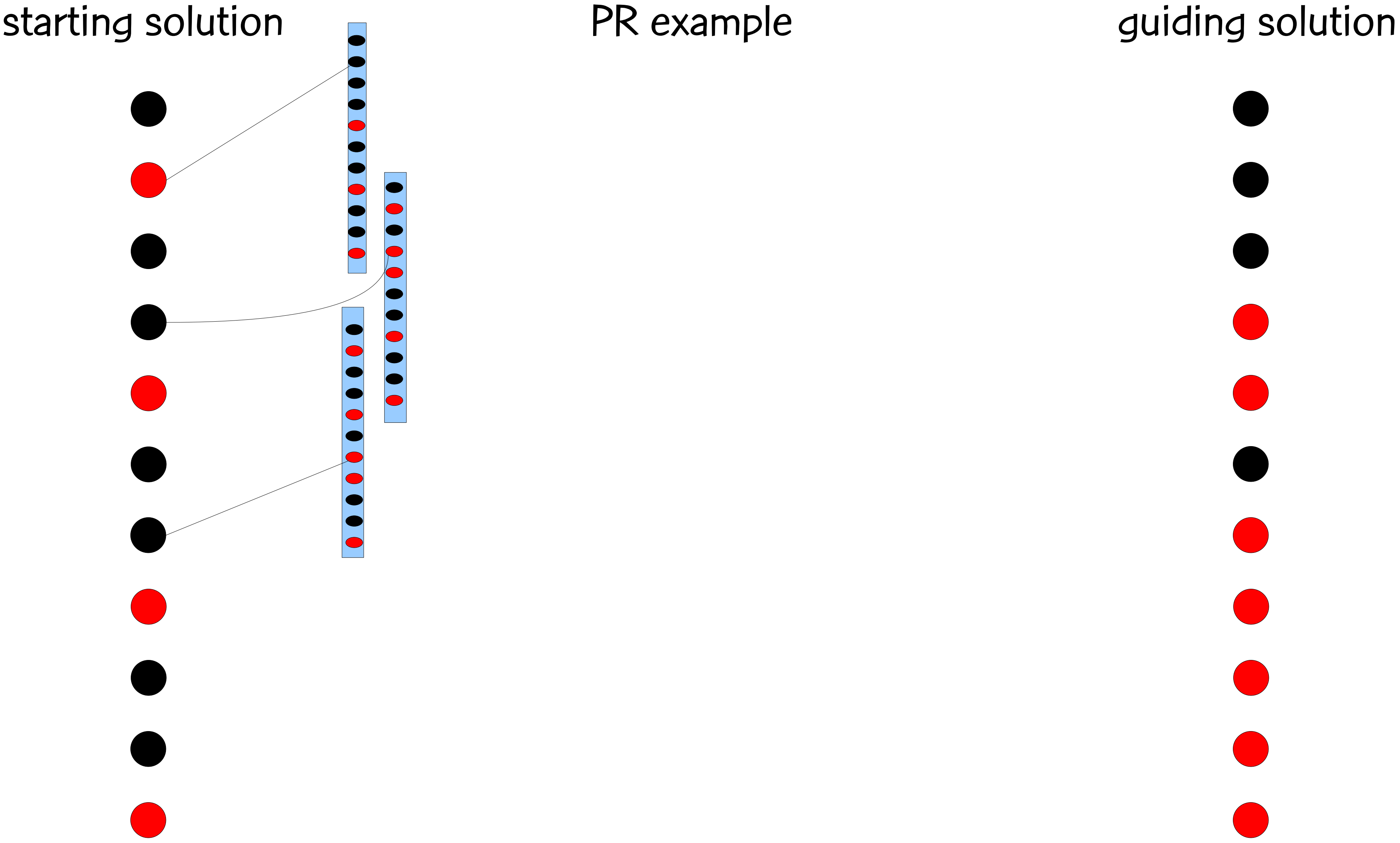
PR example



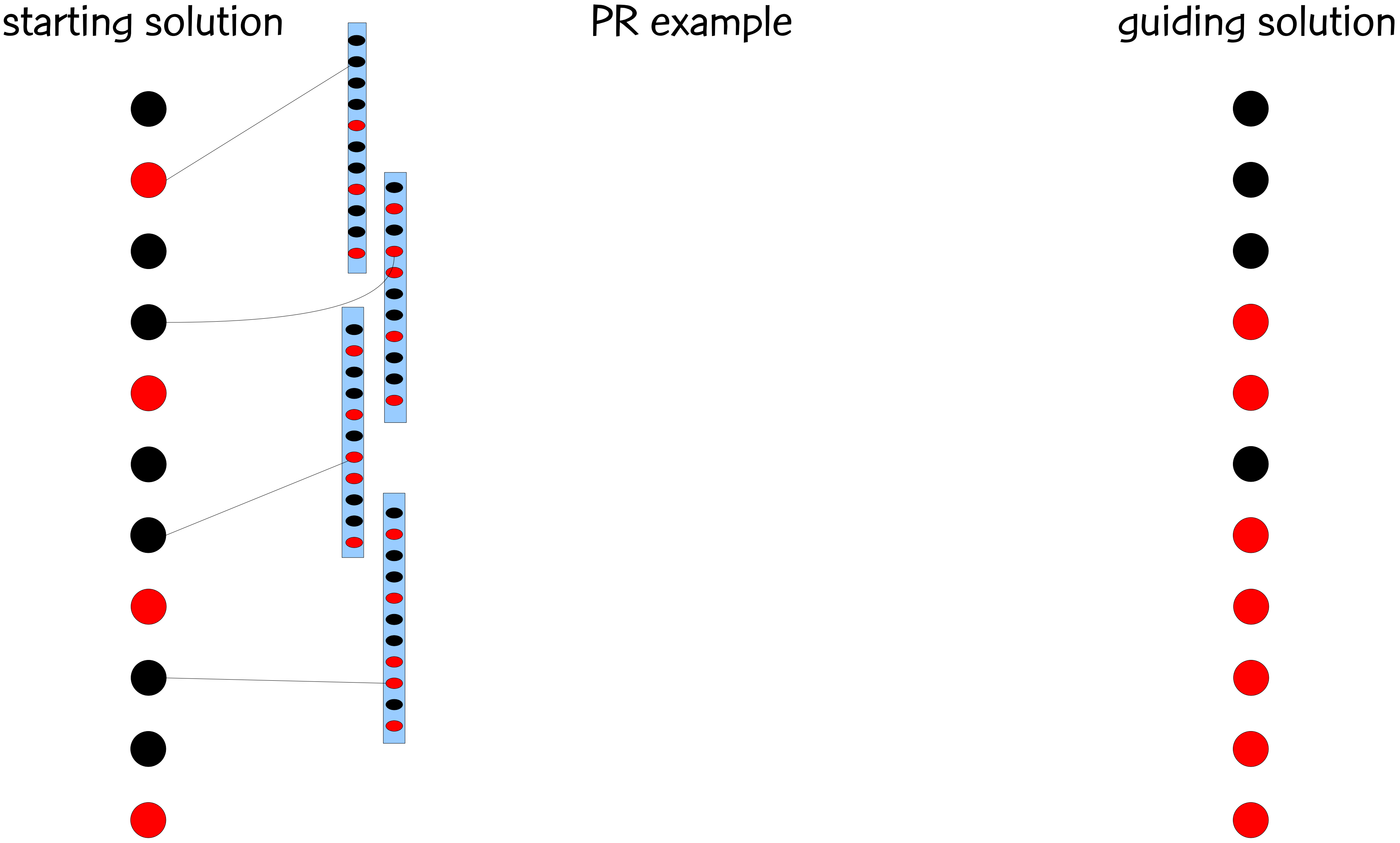
PR example



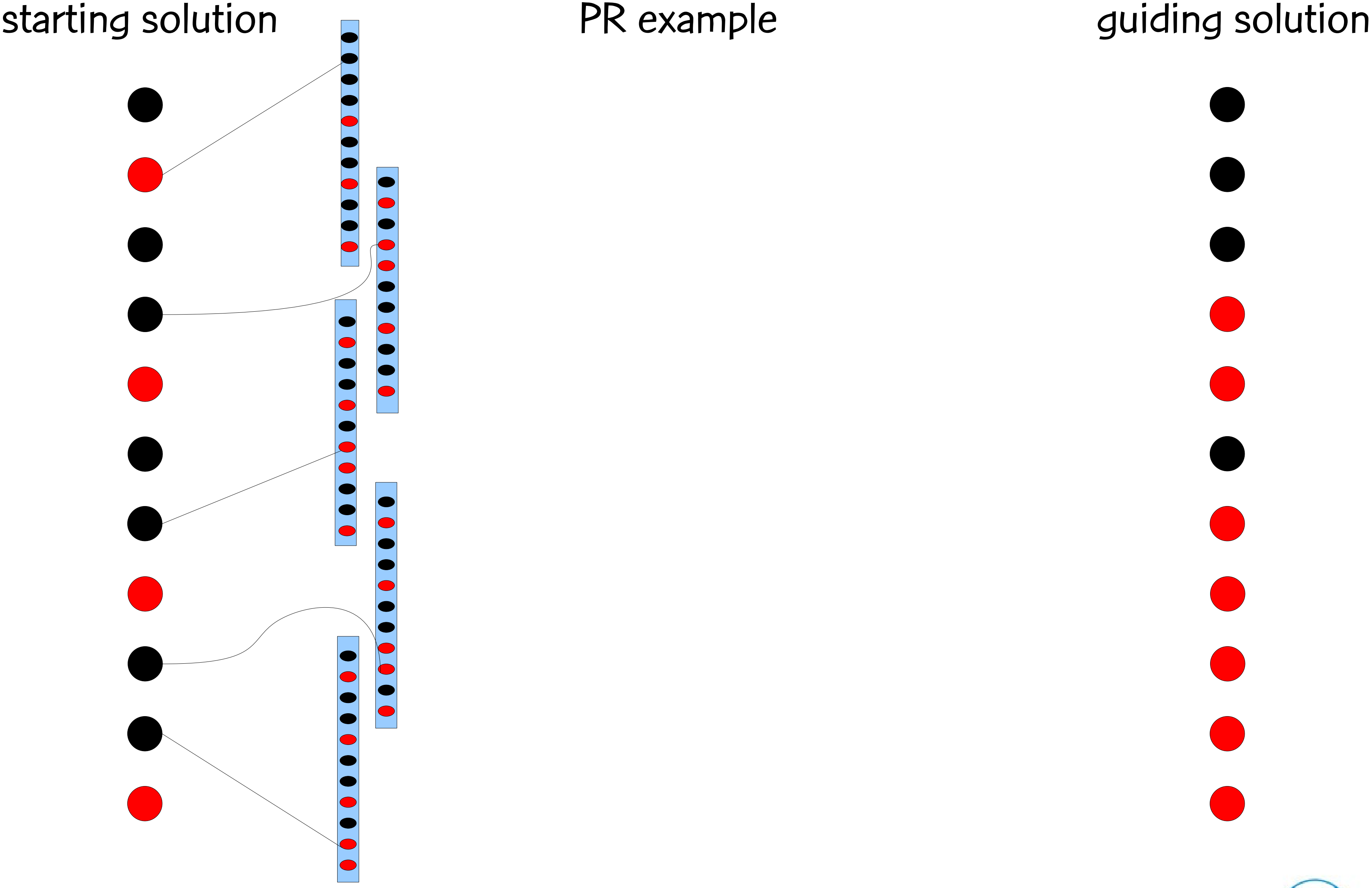
PR example



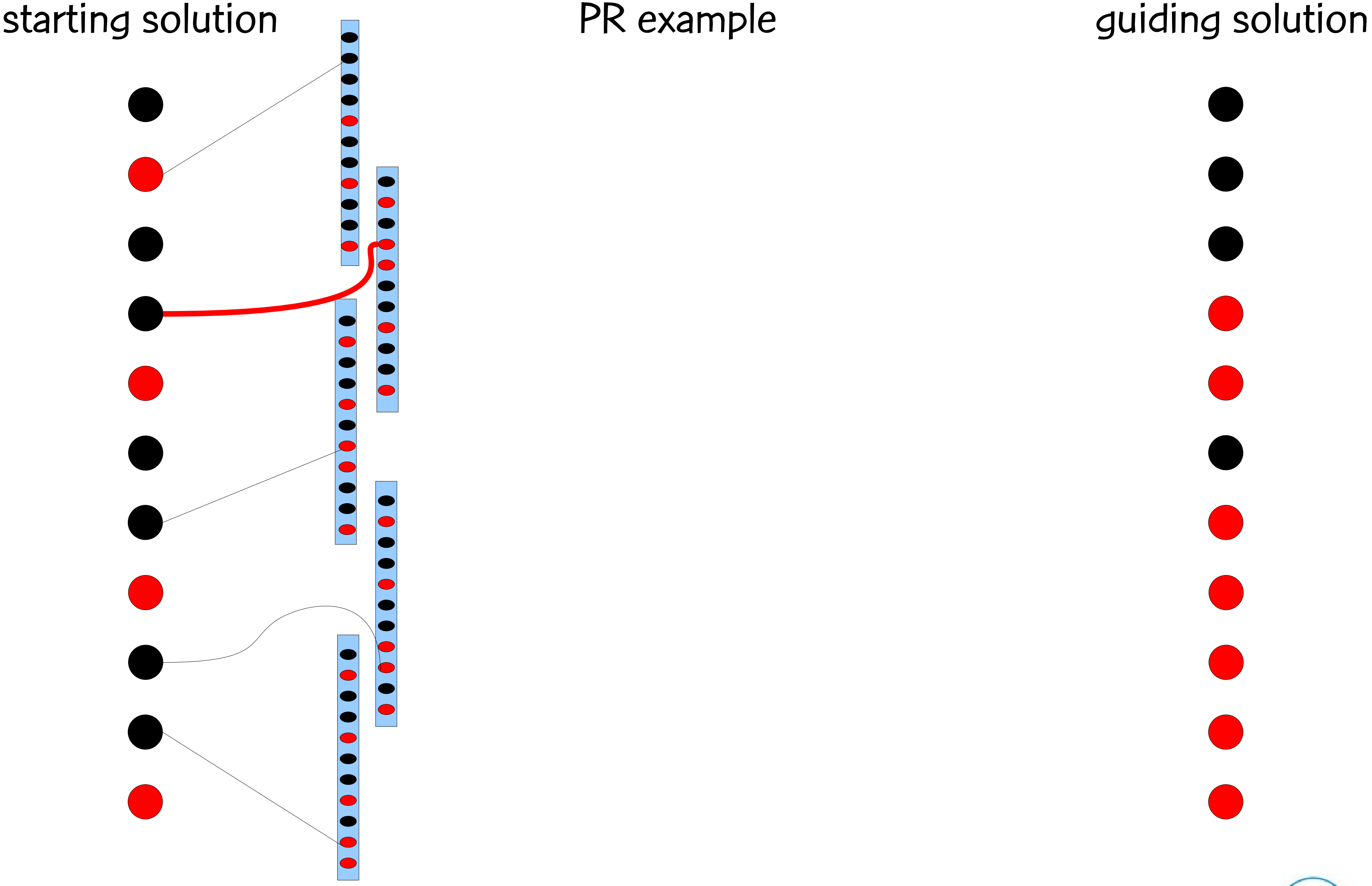
PR example



PR example

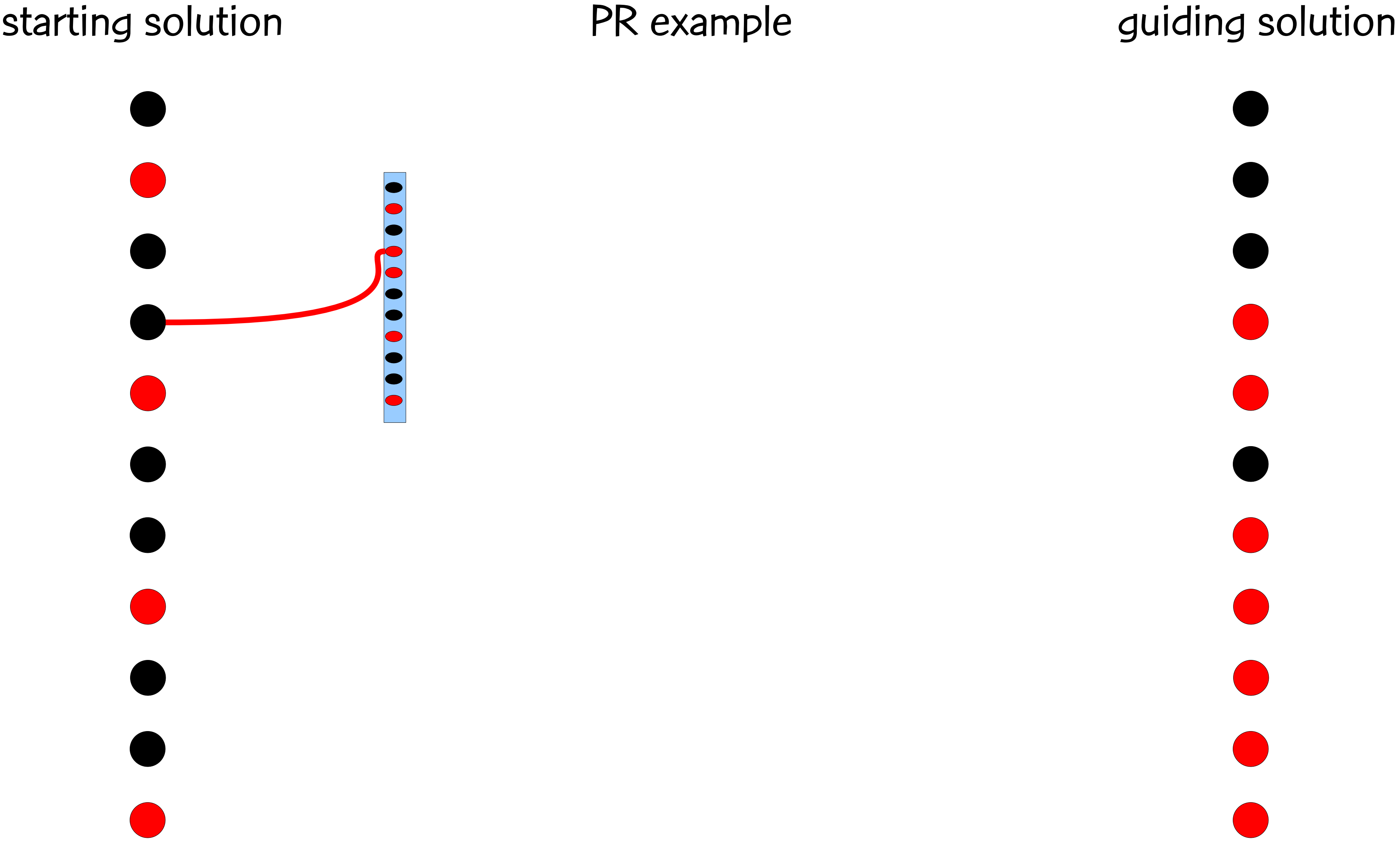


PR example

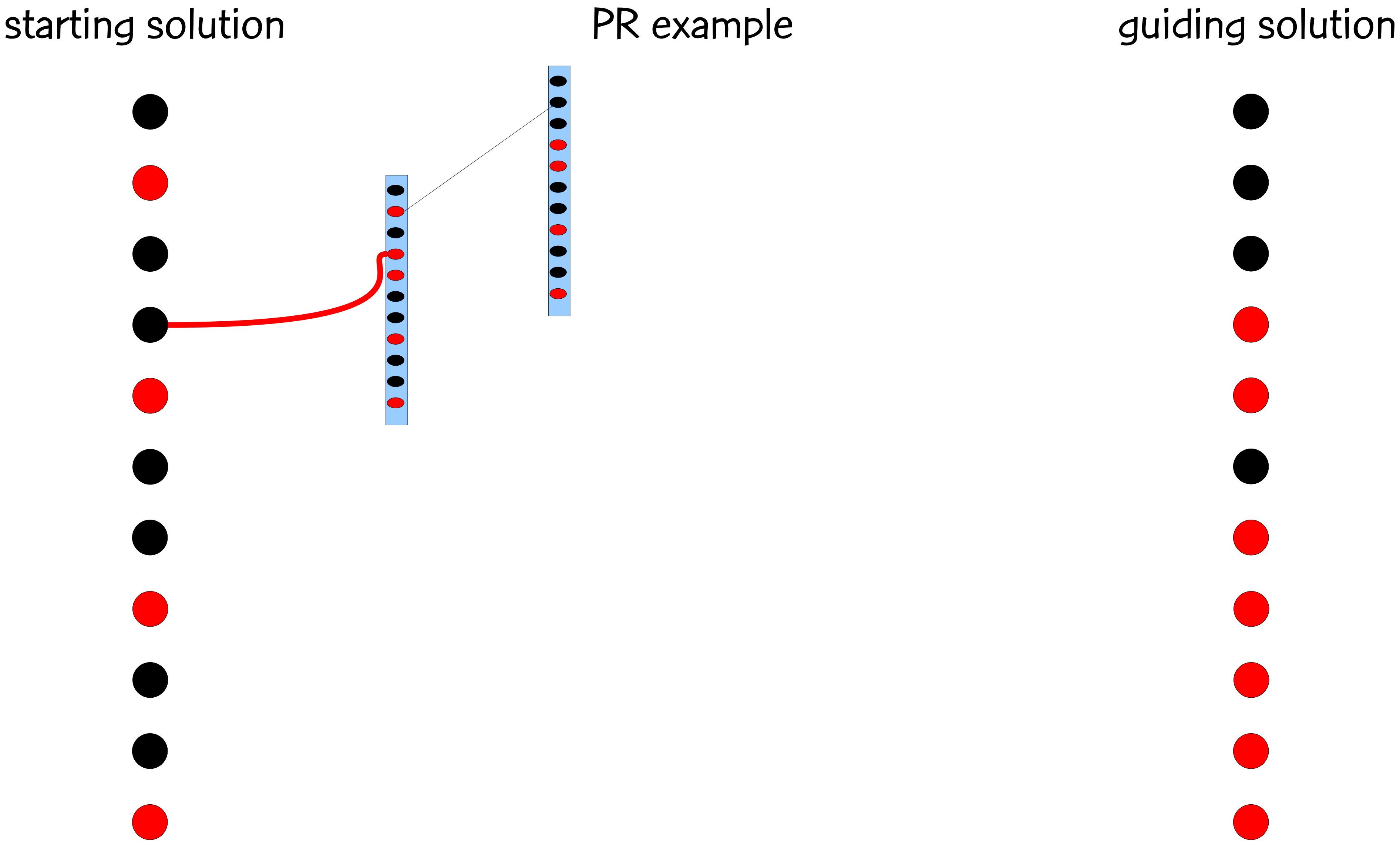


Aug. 2007

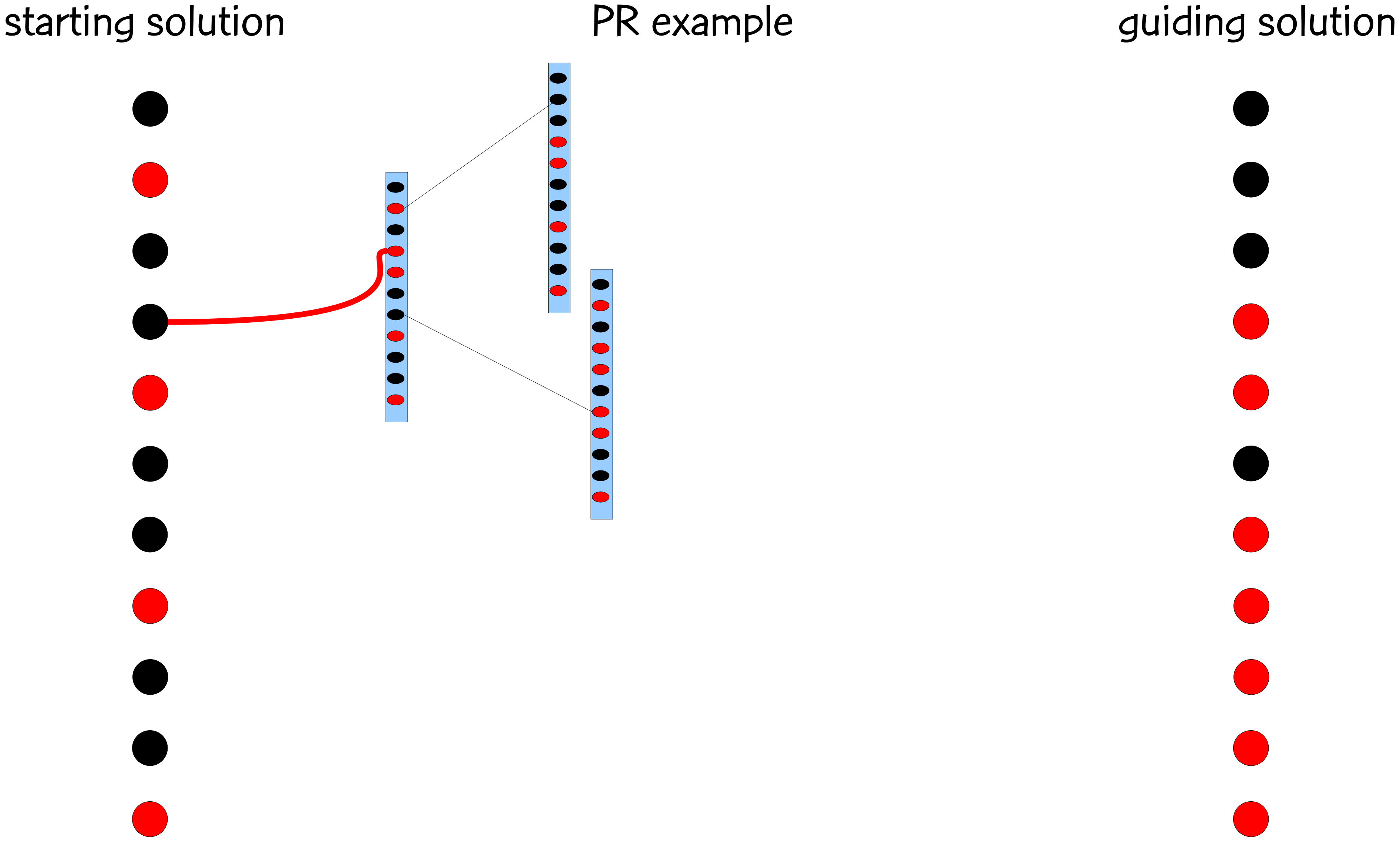
PR example



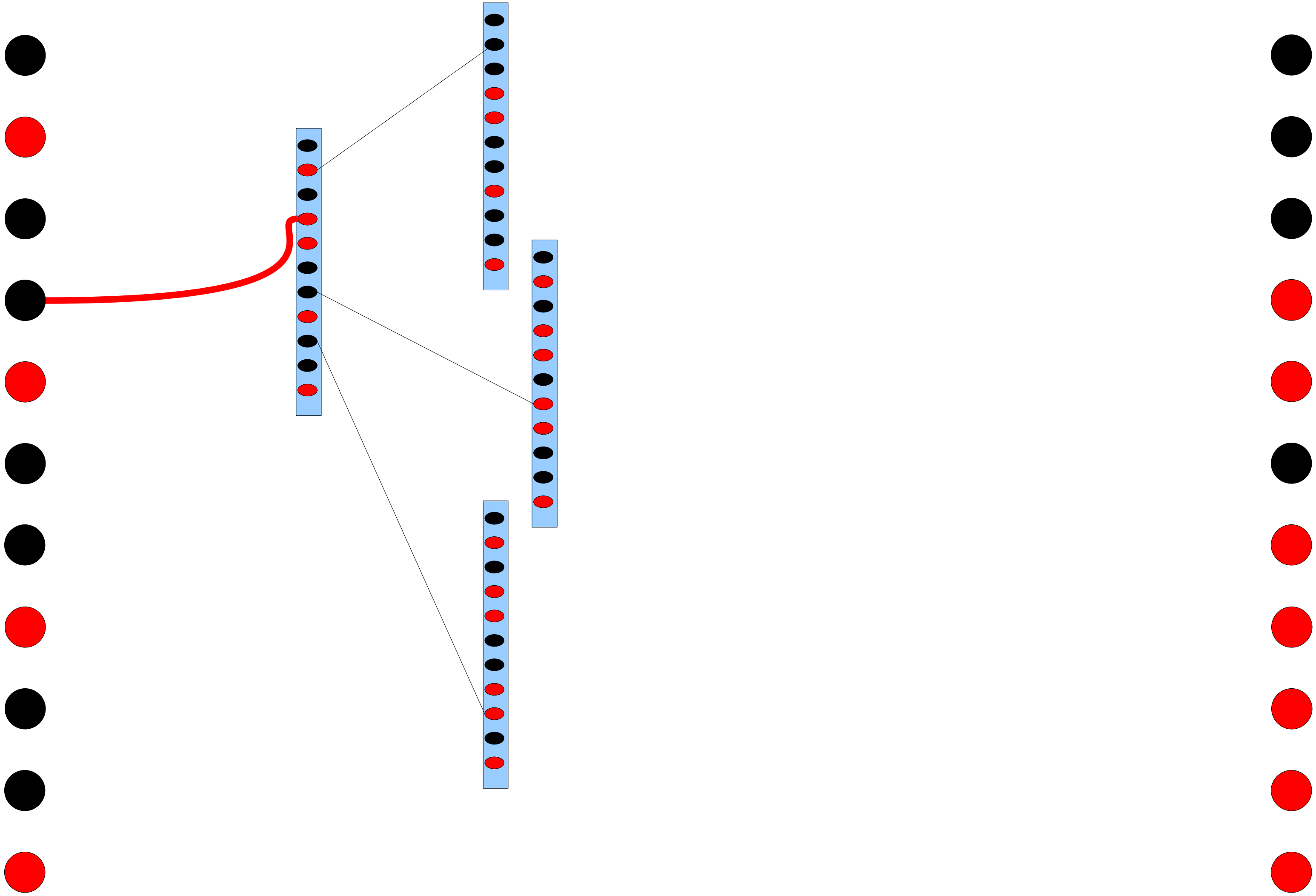
PR example



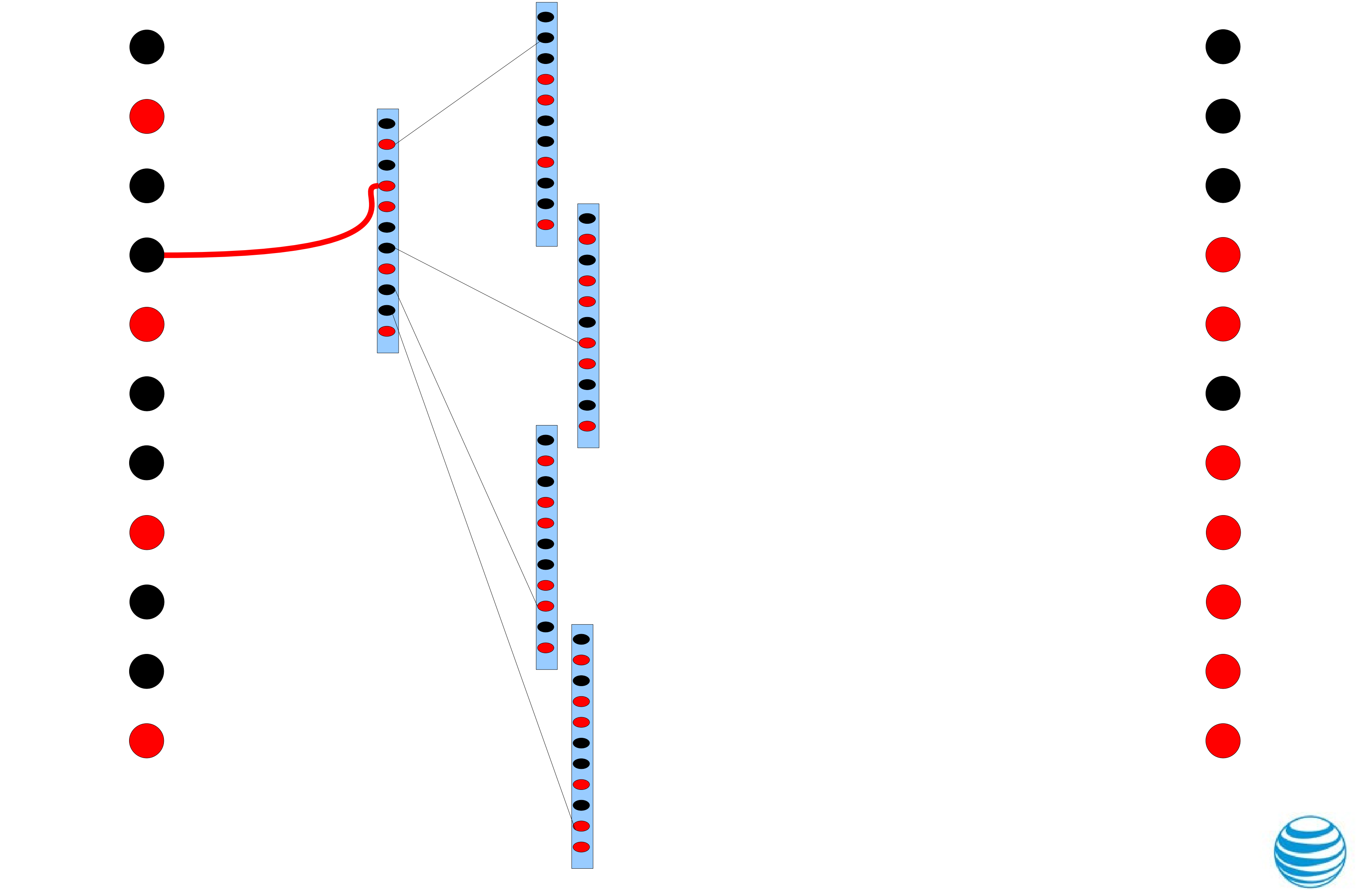
PR example



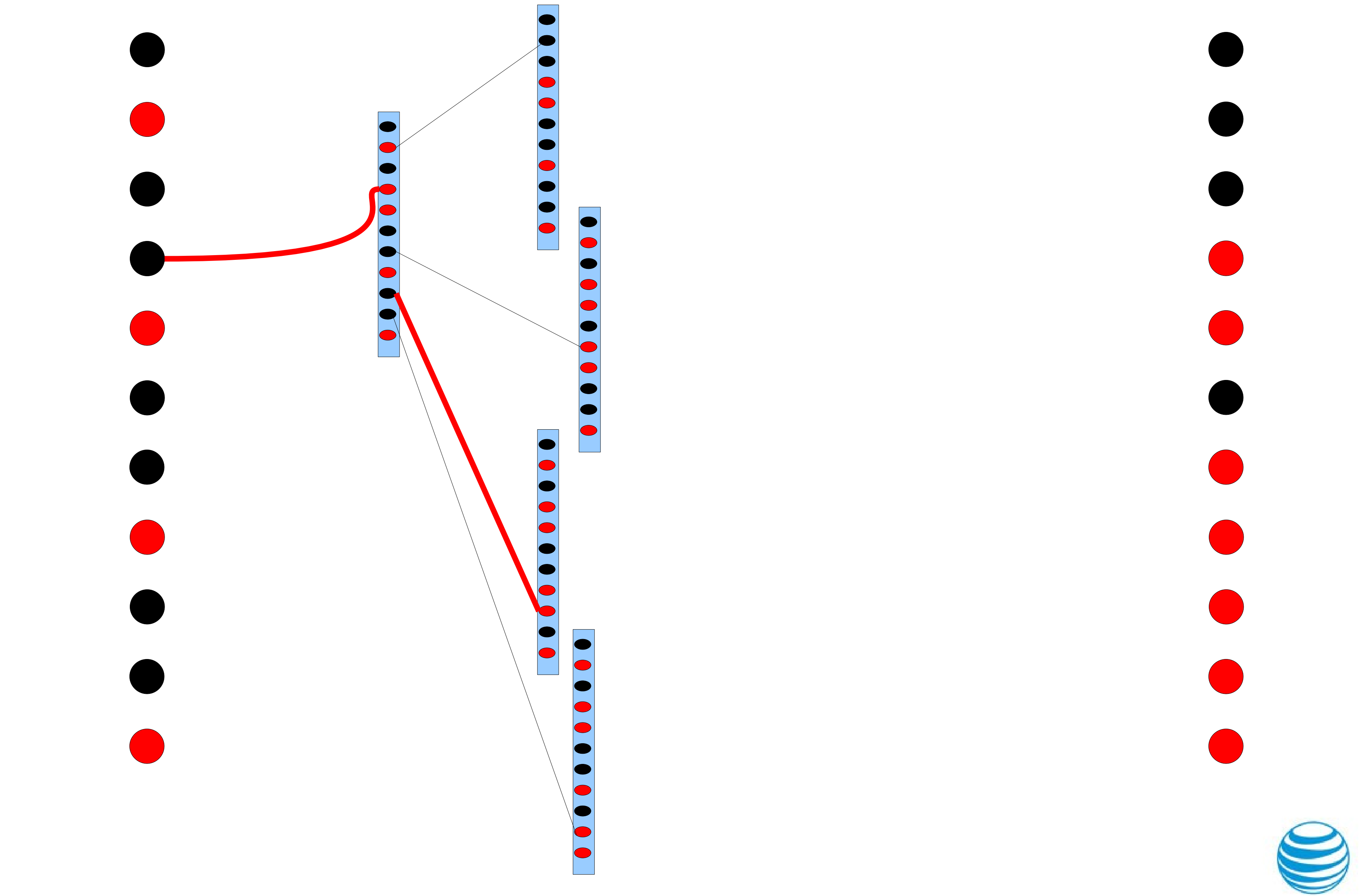
PR example



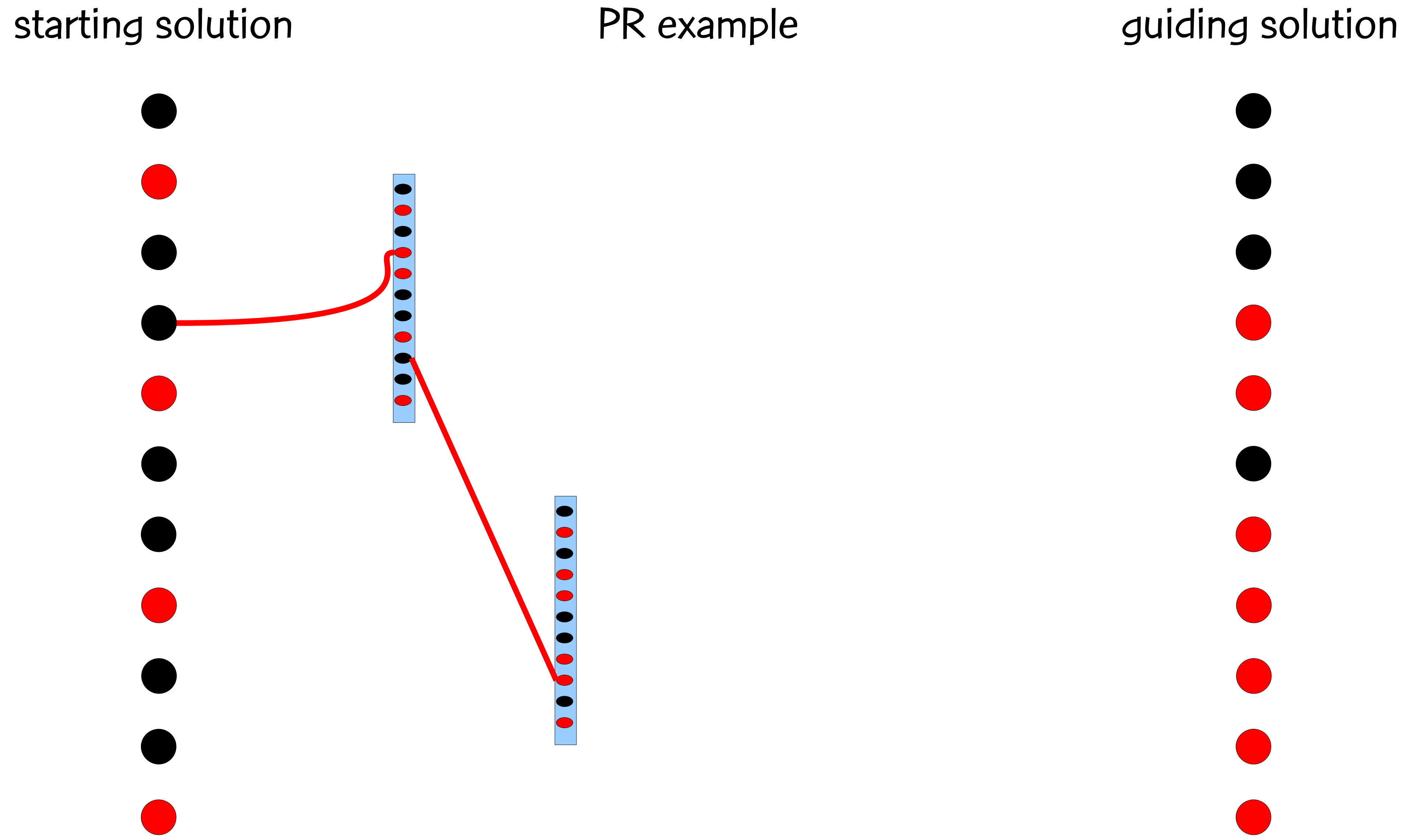
PR example



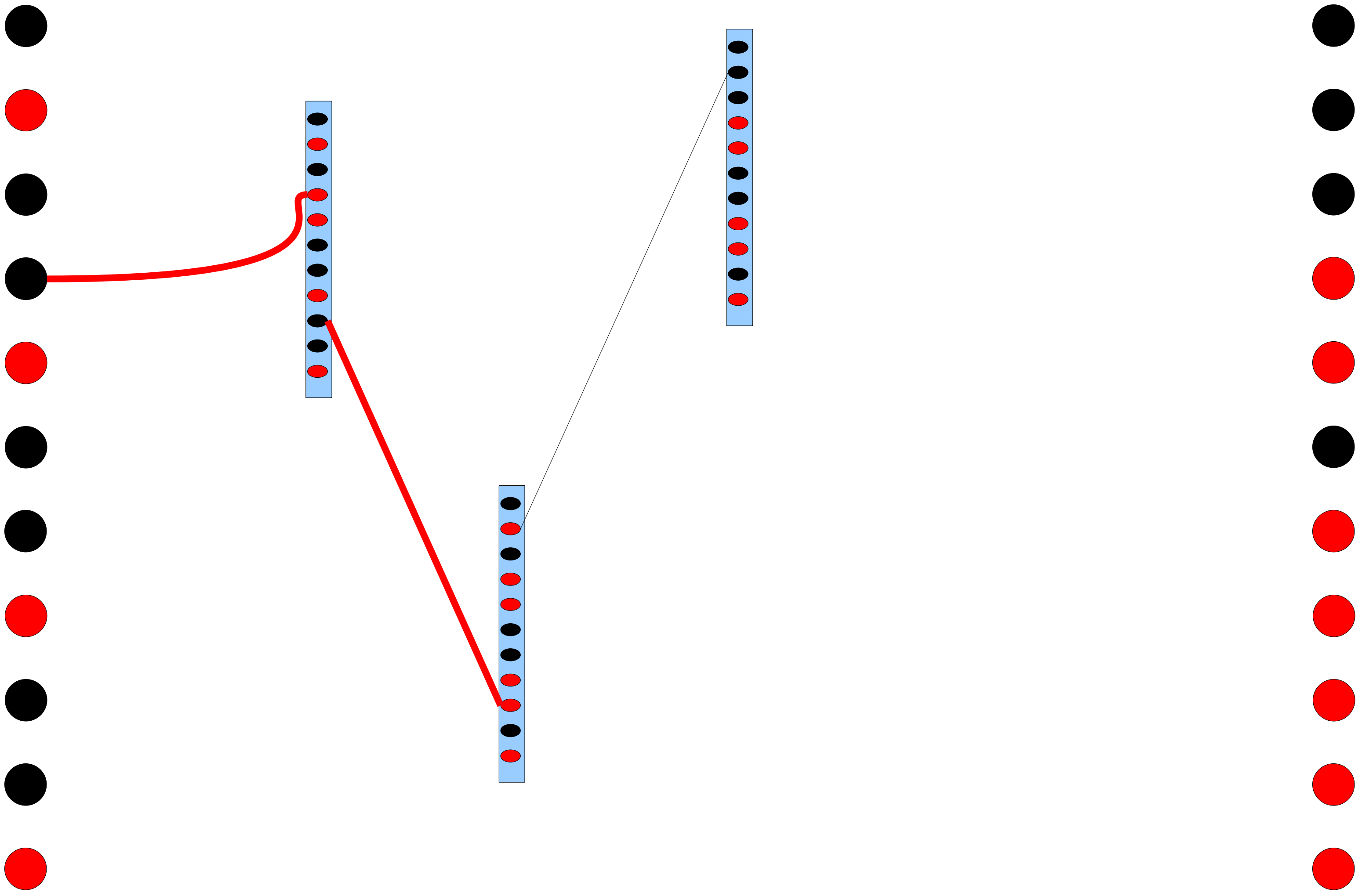
PR example



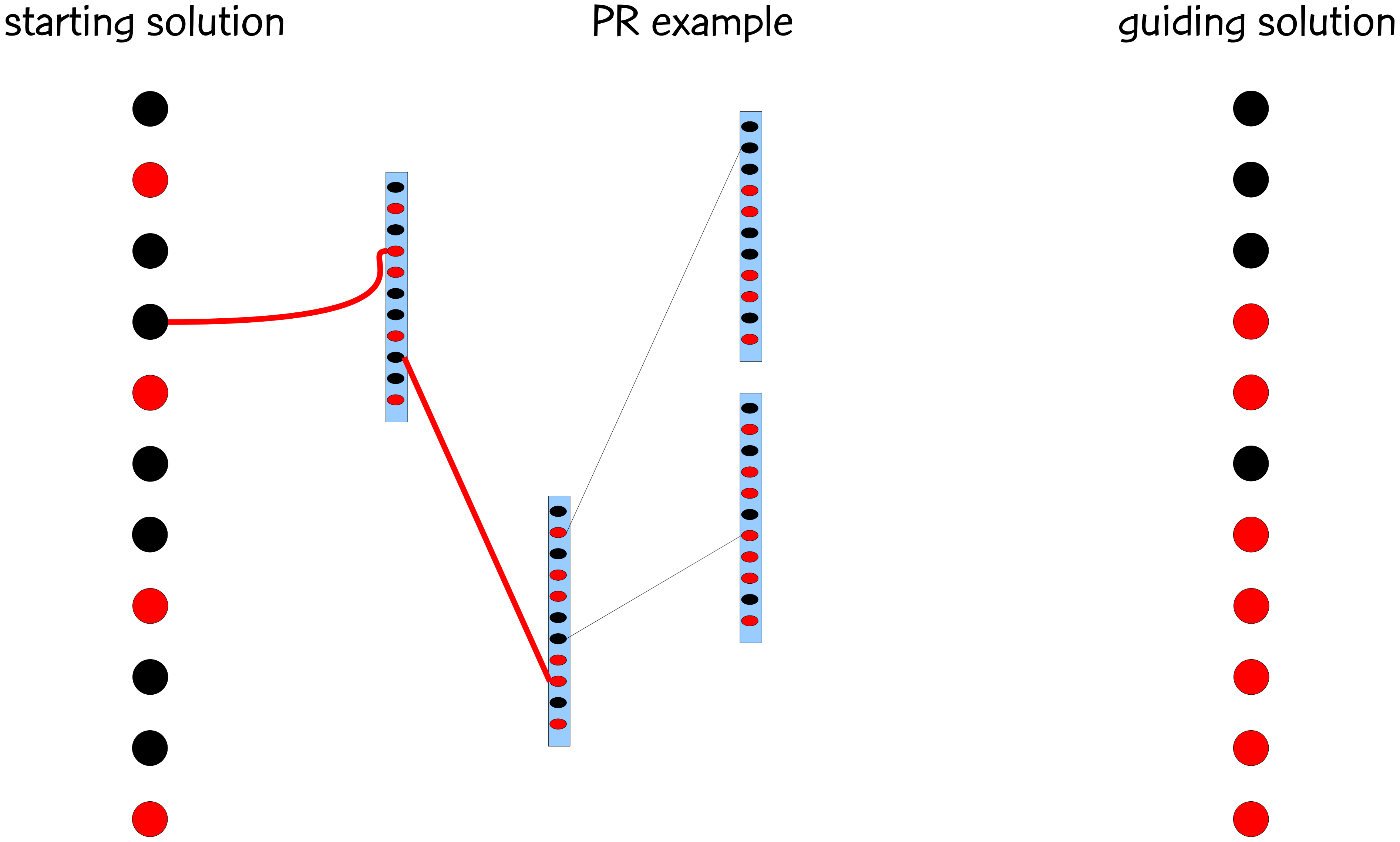
PR example



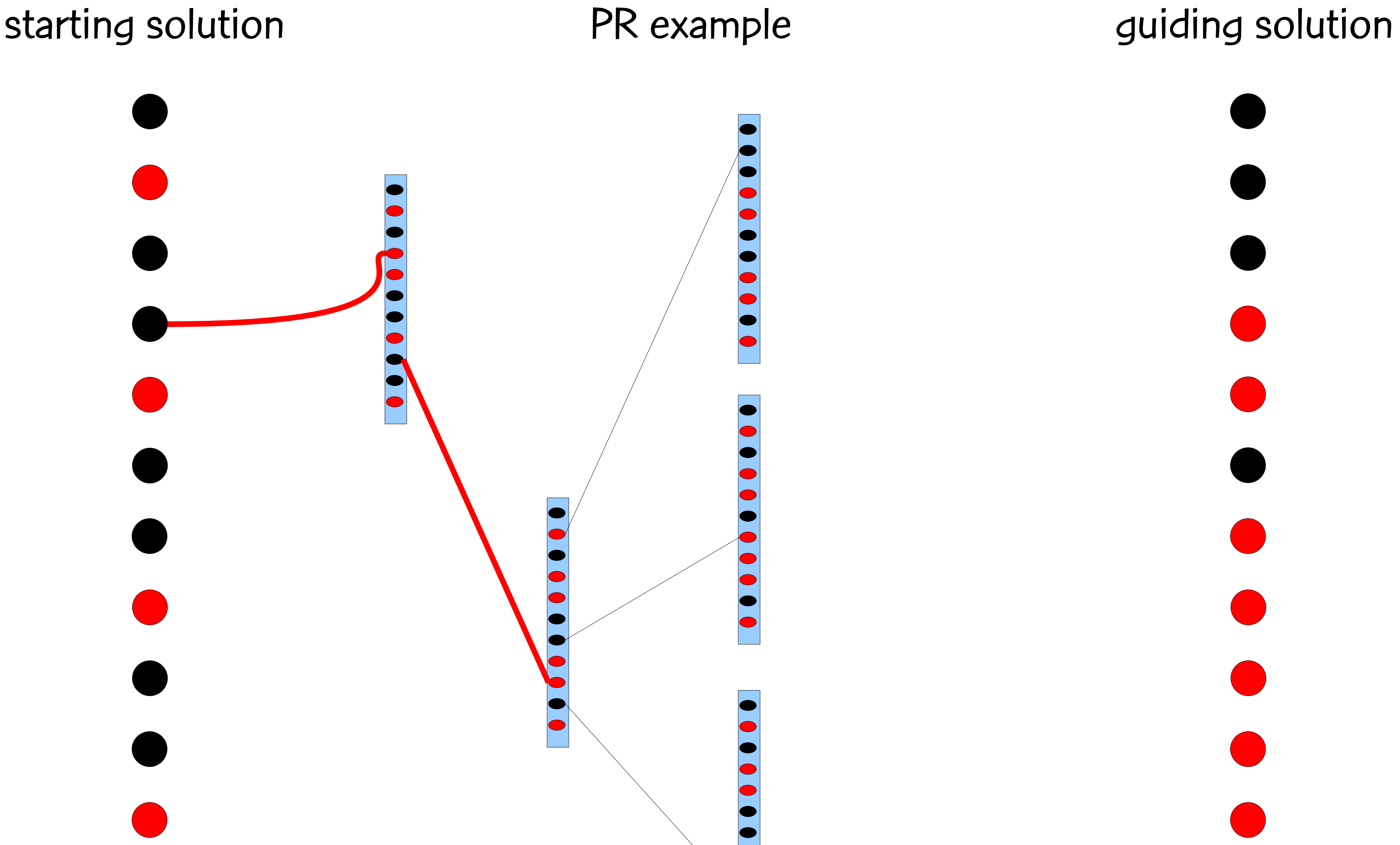
PR example



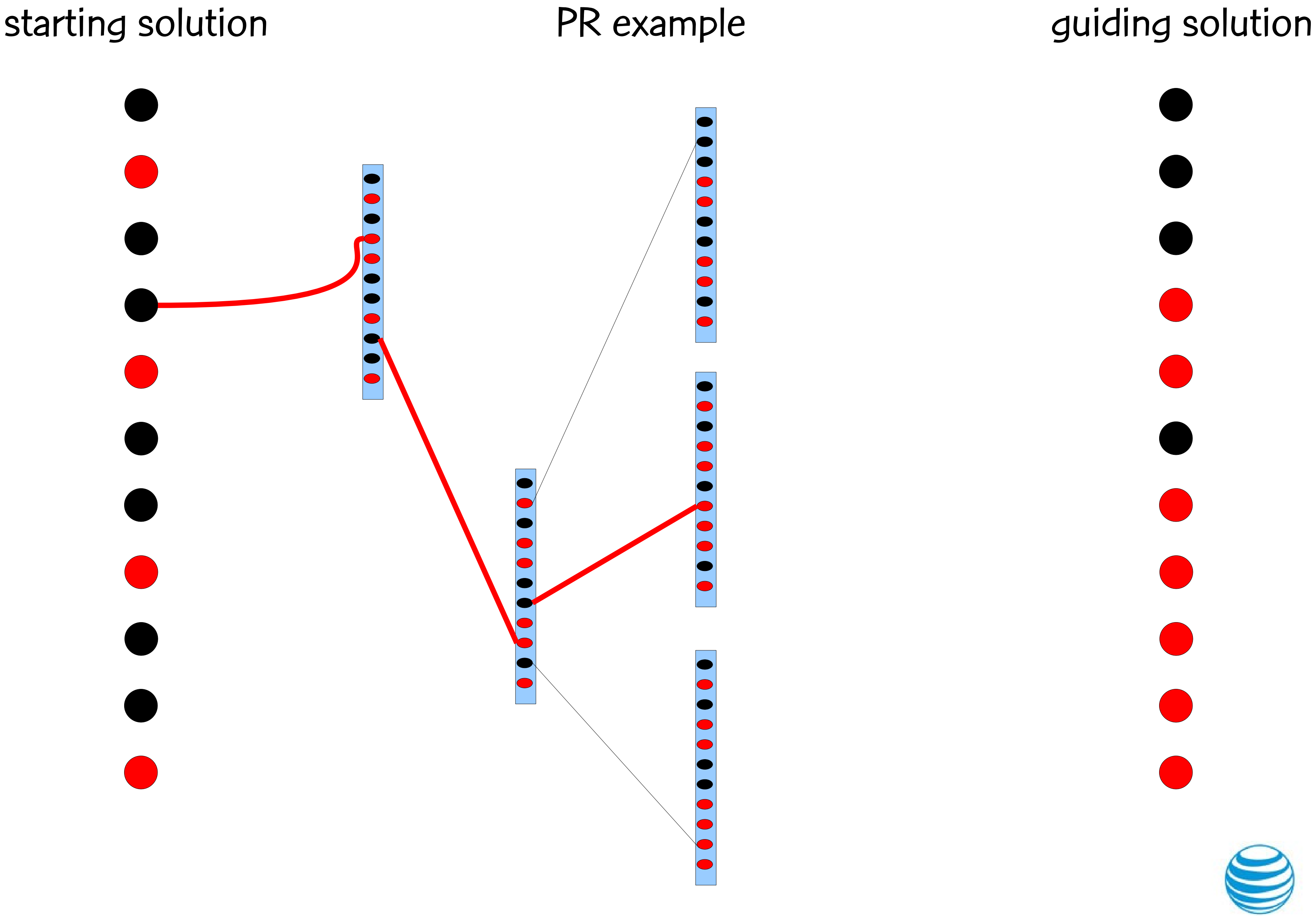
PR example



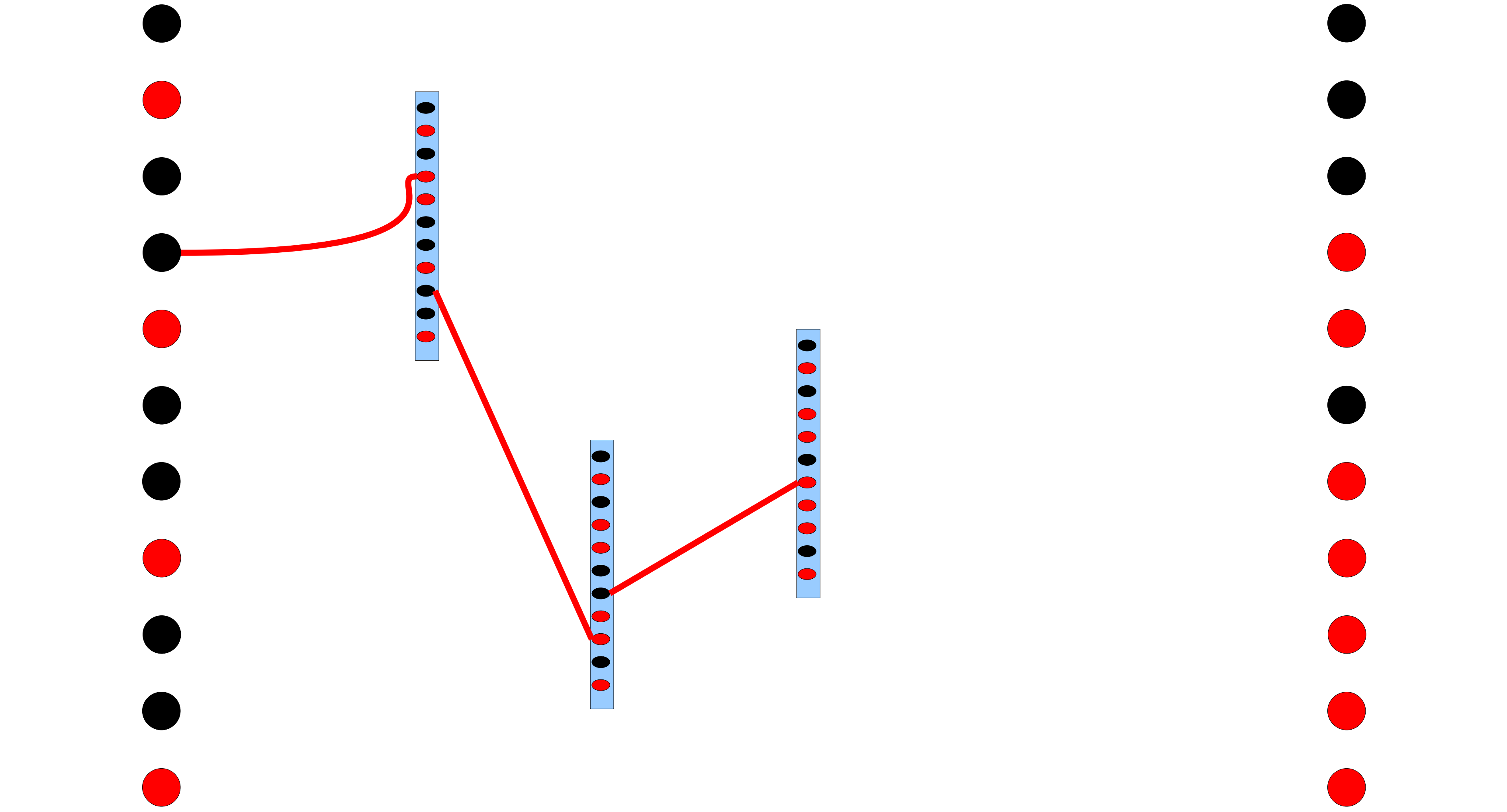
PR example



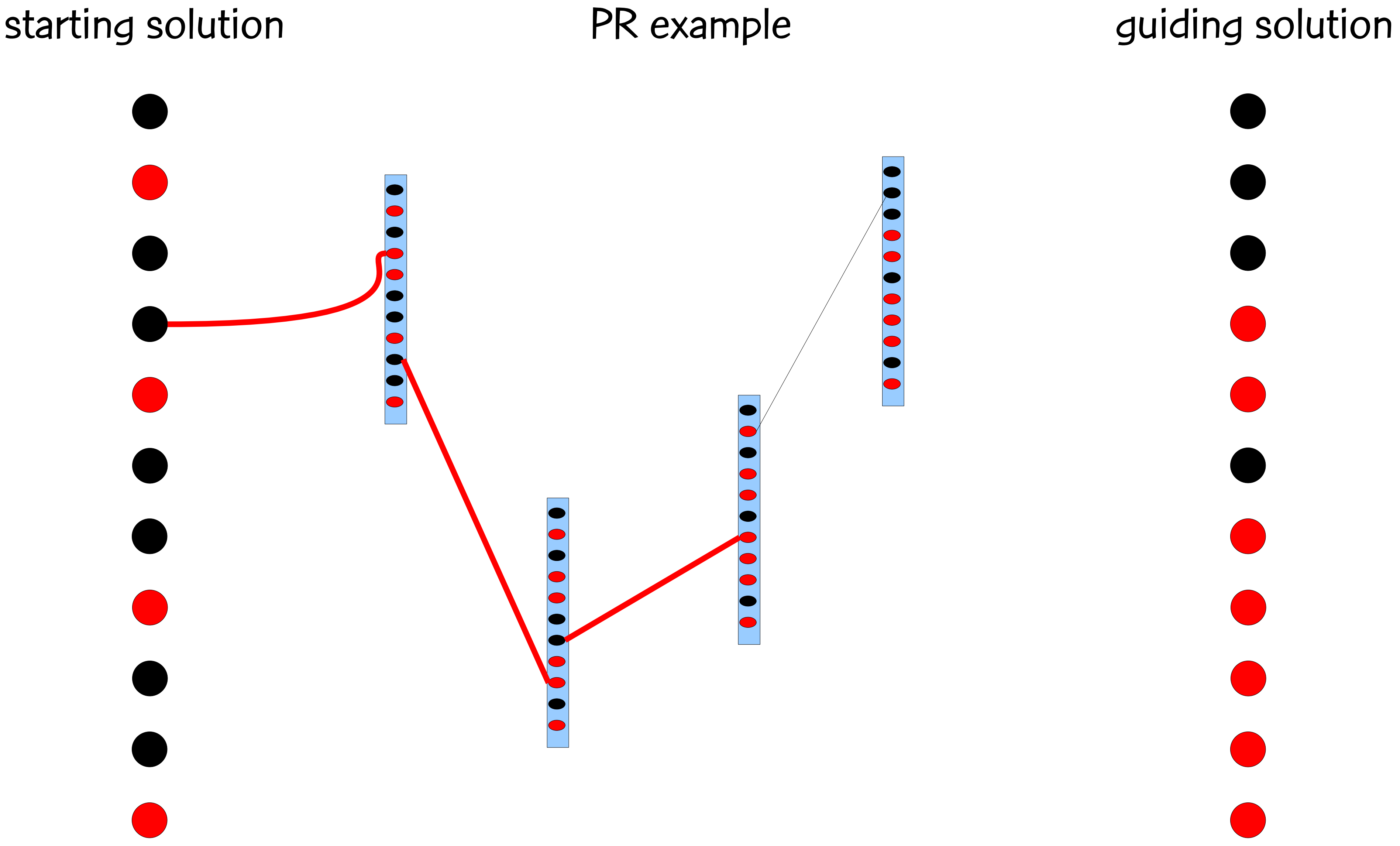
PR example



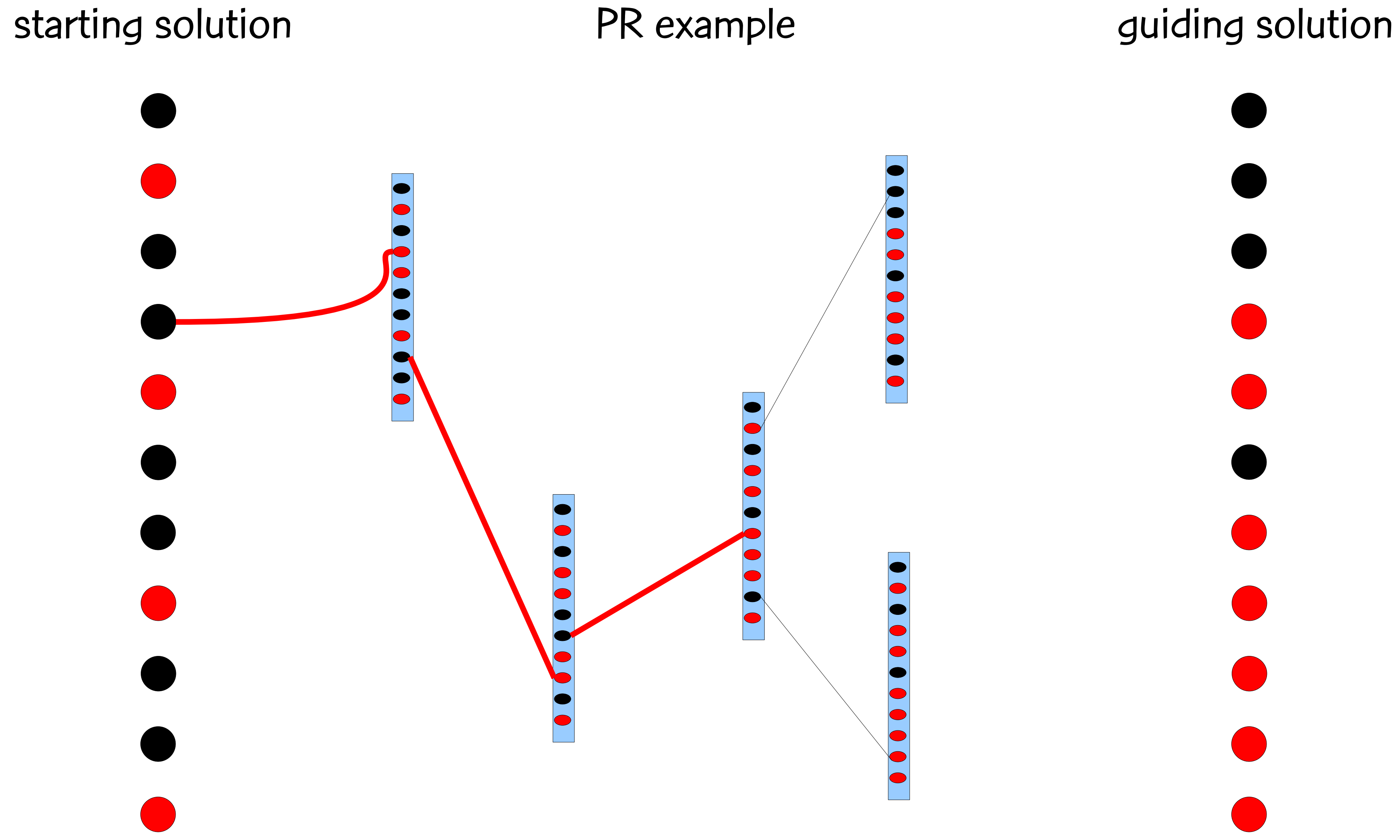
PR example



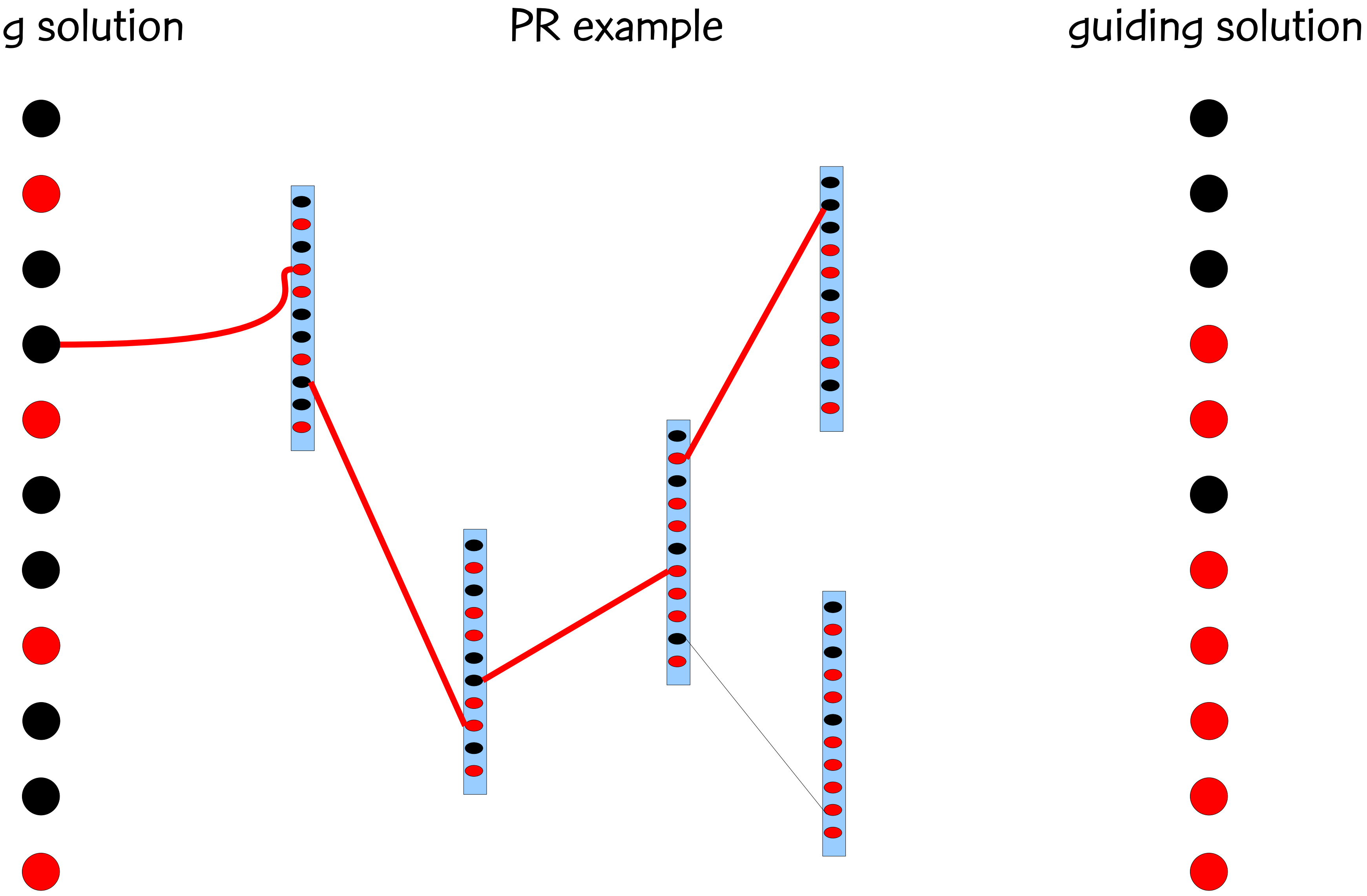
PR example



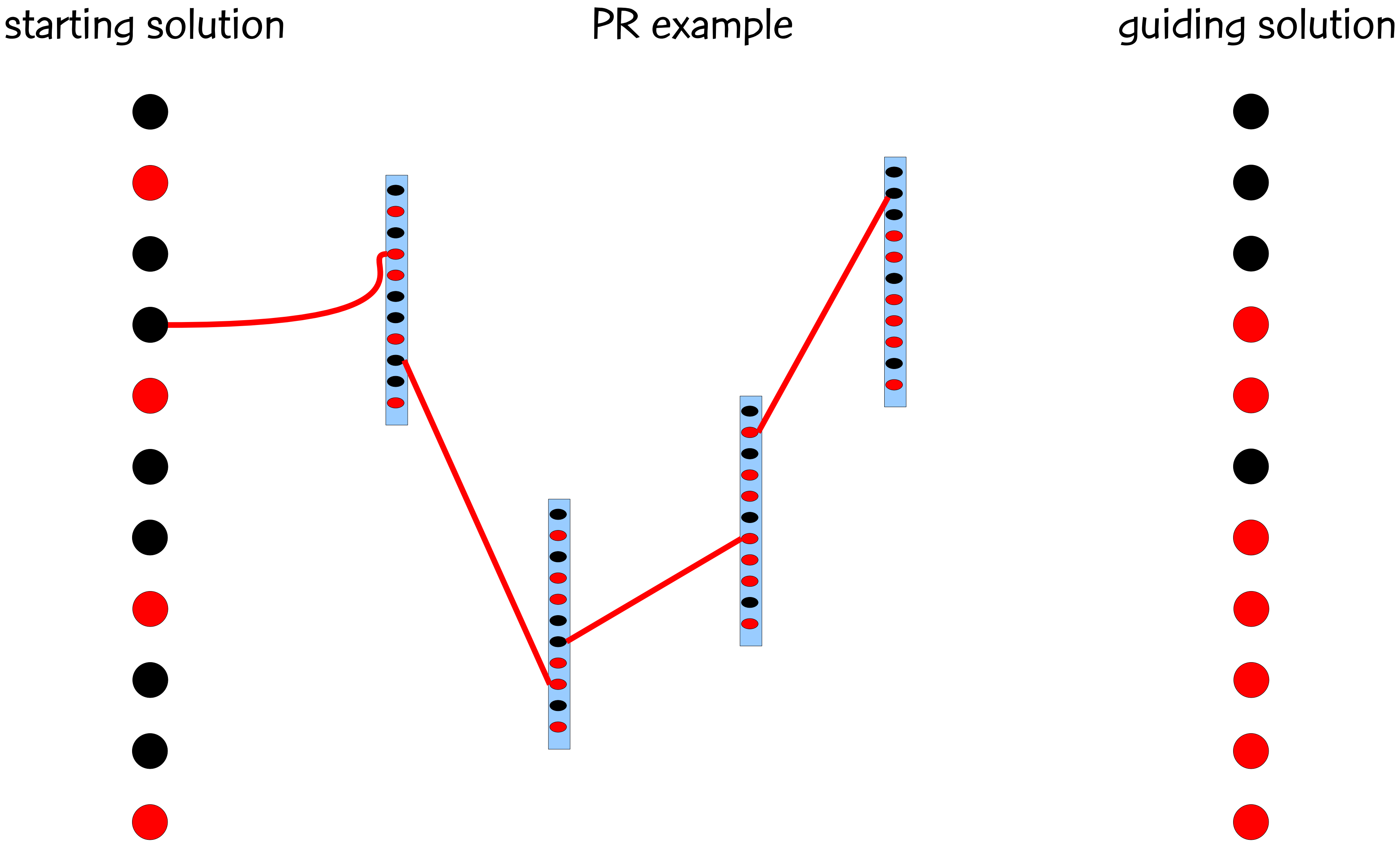
PR example



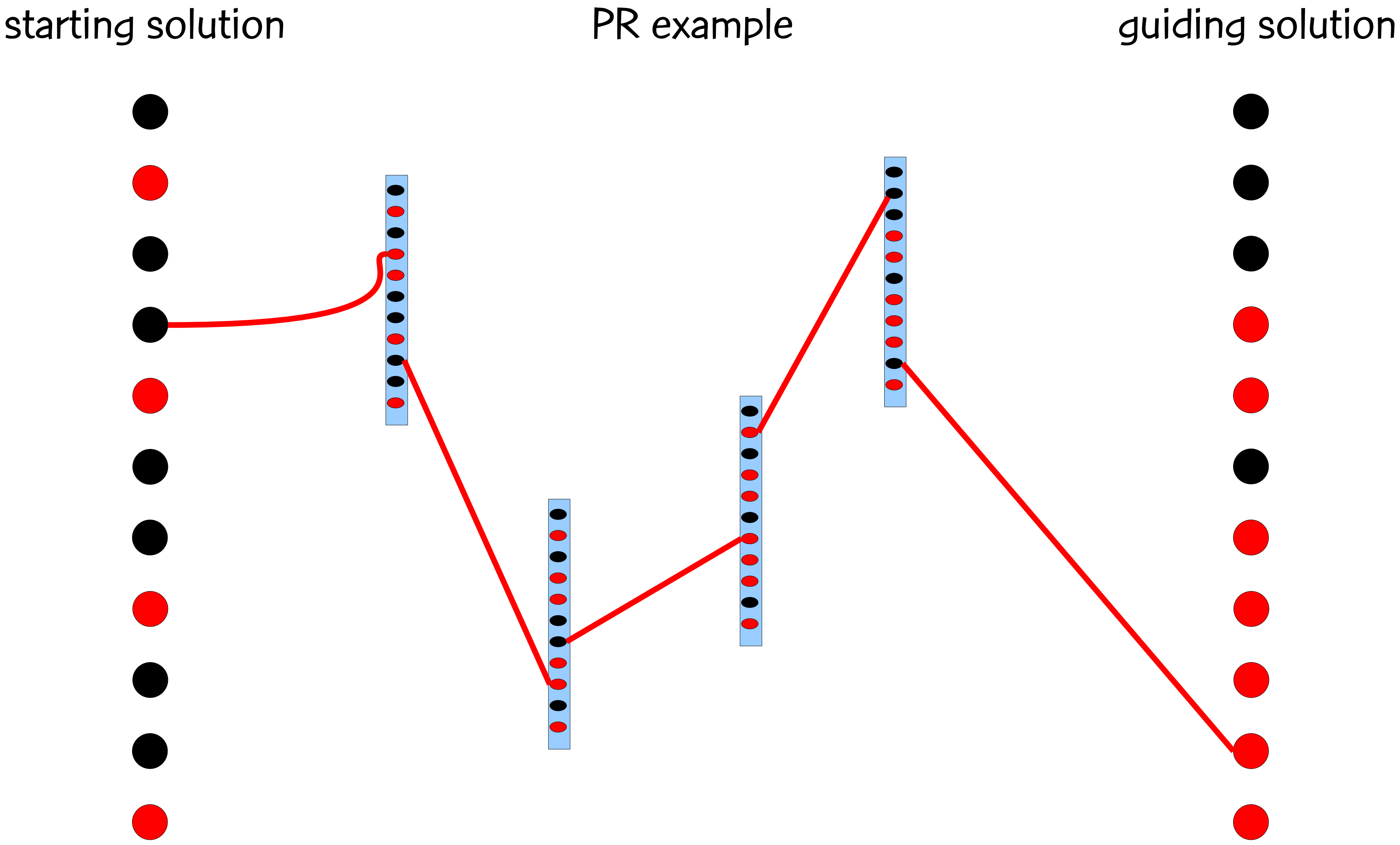
PR example



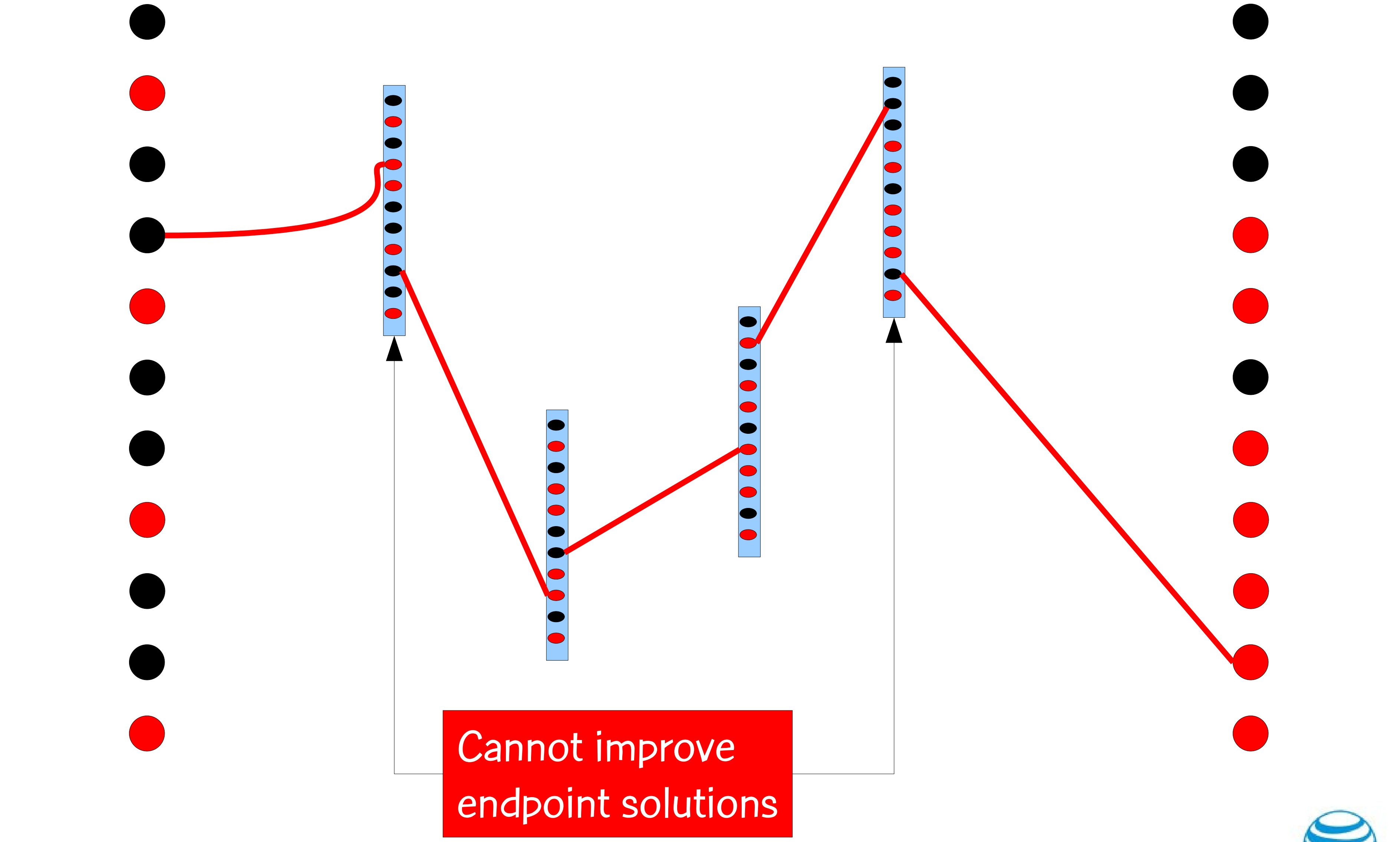
PR example



PR example

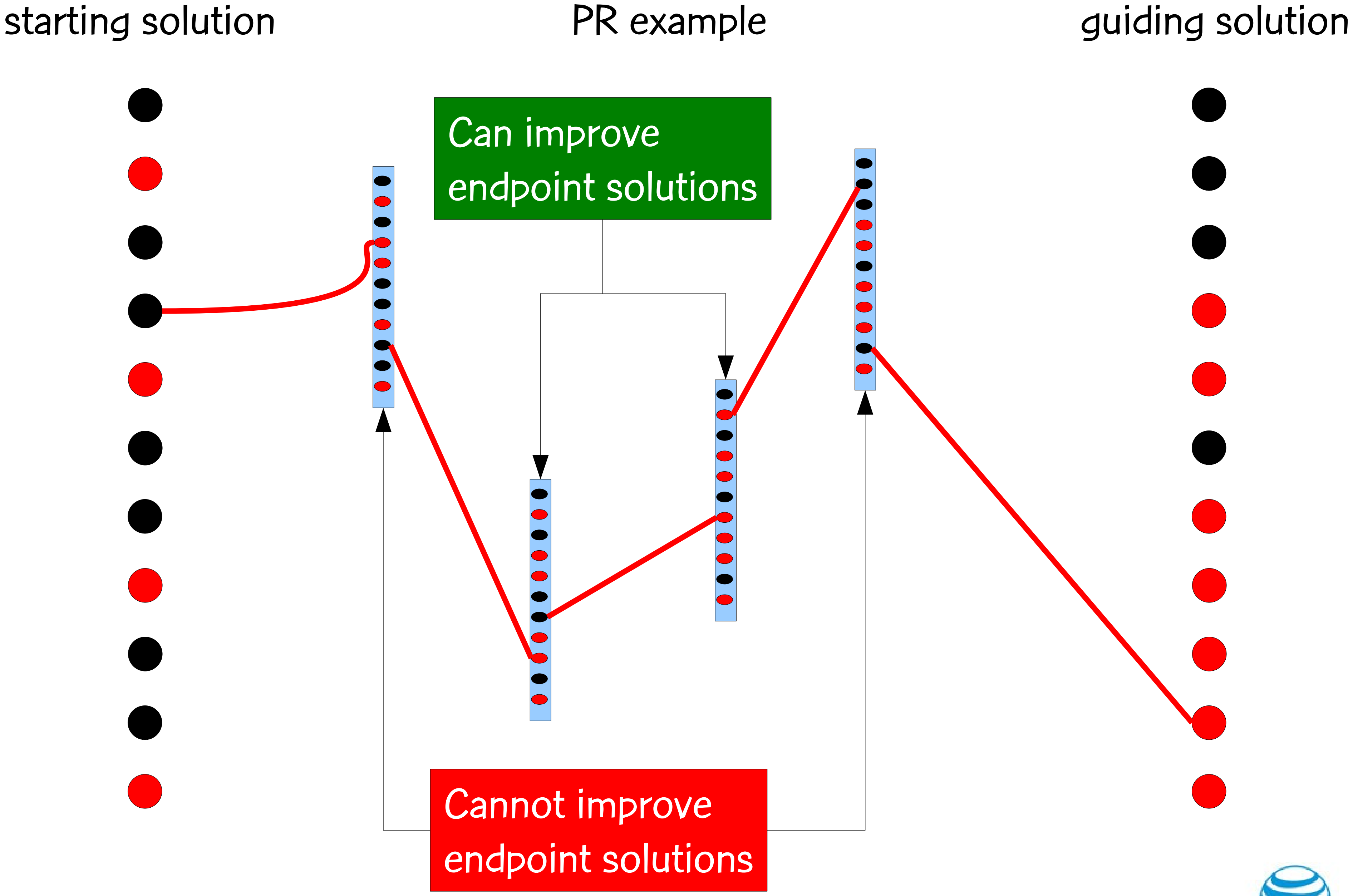


PR example



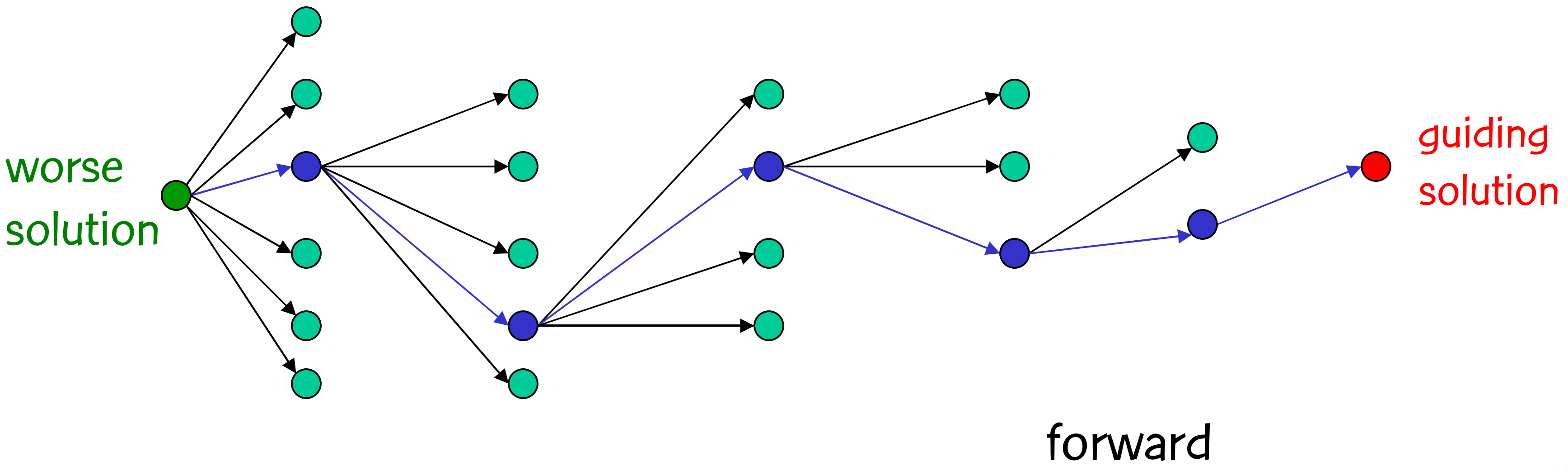
Cannot improve endpoint solutions

PR example


starting solution
guiding solution
Can improve endpoint solutions
Cannot improve endpoint solutions

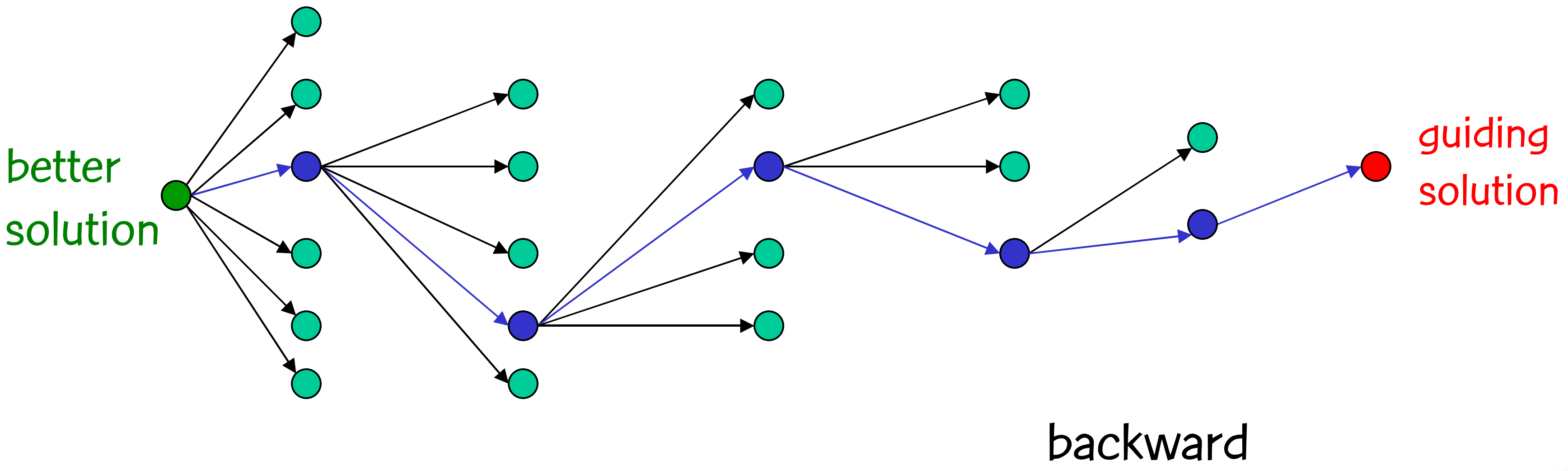
Forward path-relinking

- Variants: trade-offs between computation time and solution quality
  - Forward PR adopts as initial solution the worse of the two input solutions and uses the better solution as the guide.



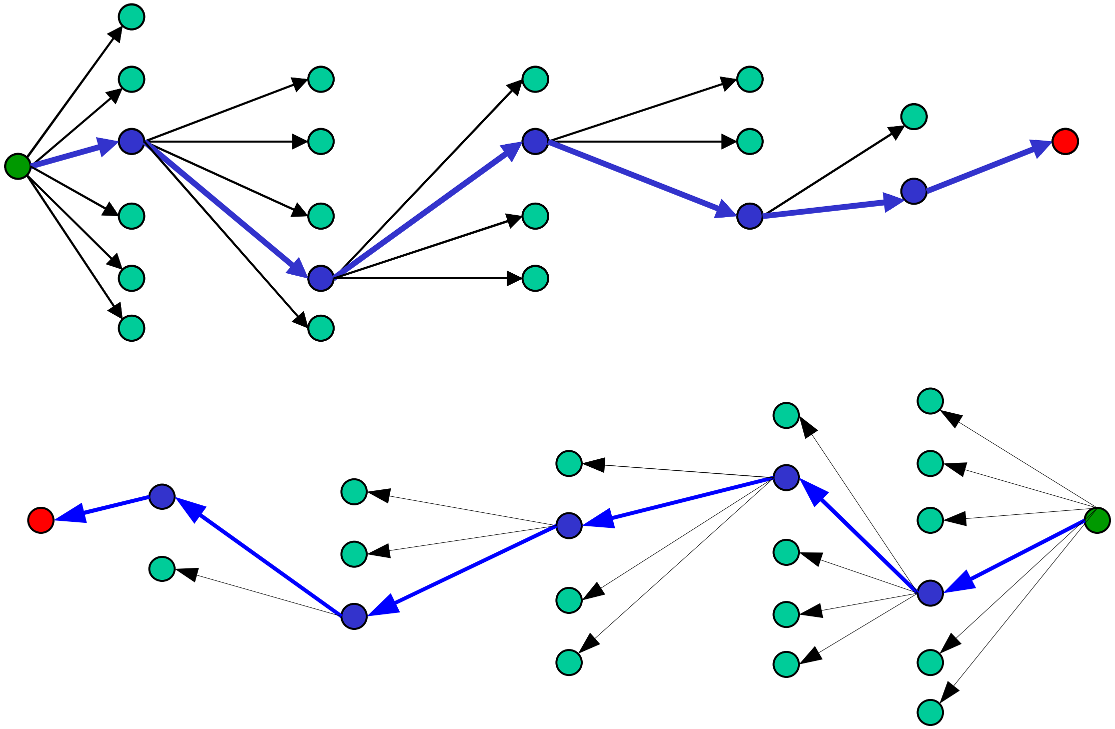
Backward path-relinking

- Variants: trade-offs between computation time and solution quality
  - Backward PR usually does better: Better to start from the best of the two input solutions, neighborhood of the initial solution is explored more than of the guide!



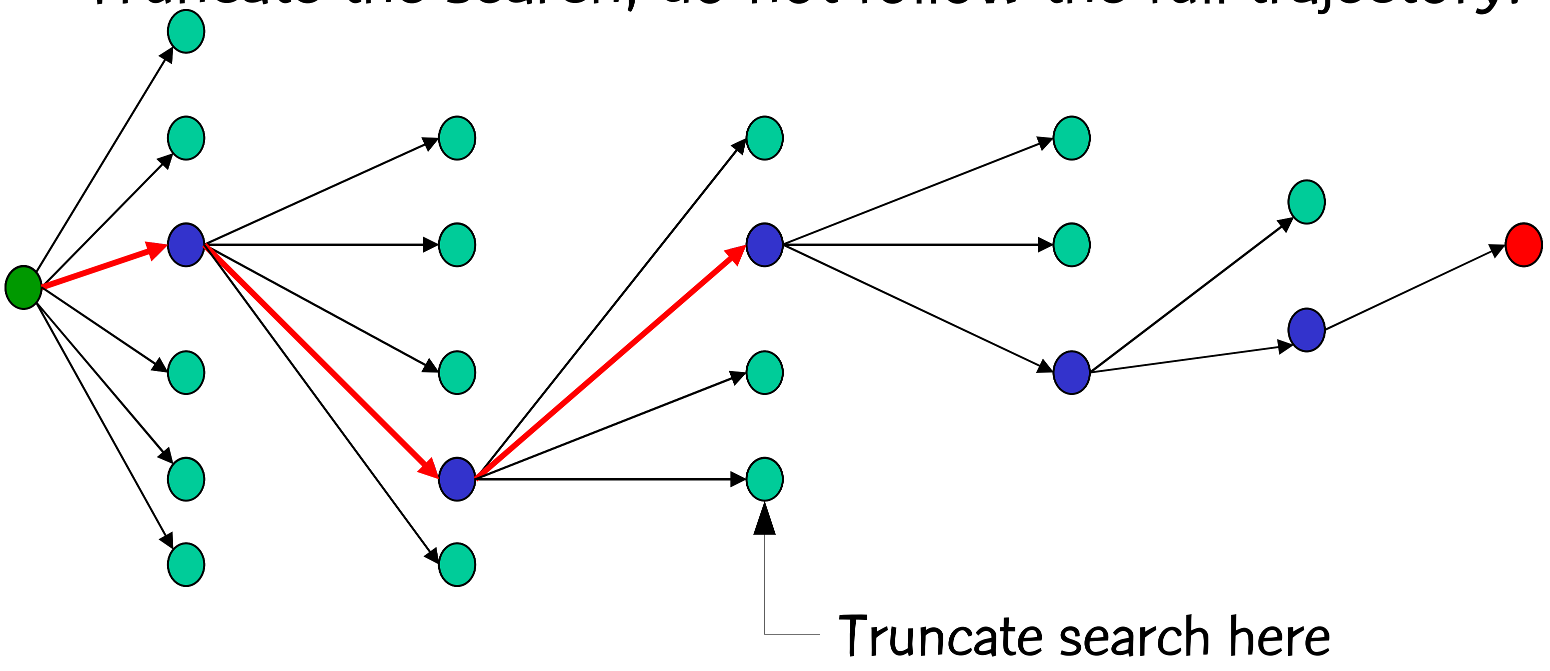
Back and forth path-relinking

- Variants: trade-offs between computation time and solution quality
  - Explore both trajectories: twice as much time, often with only marginal improvements!



Truncated path-relinking

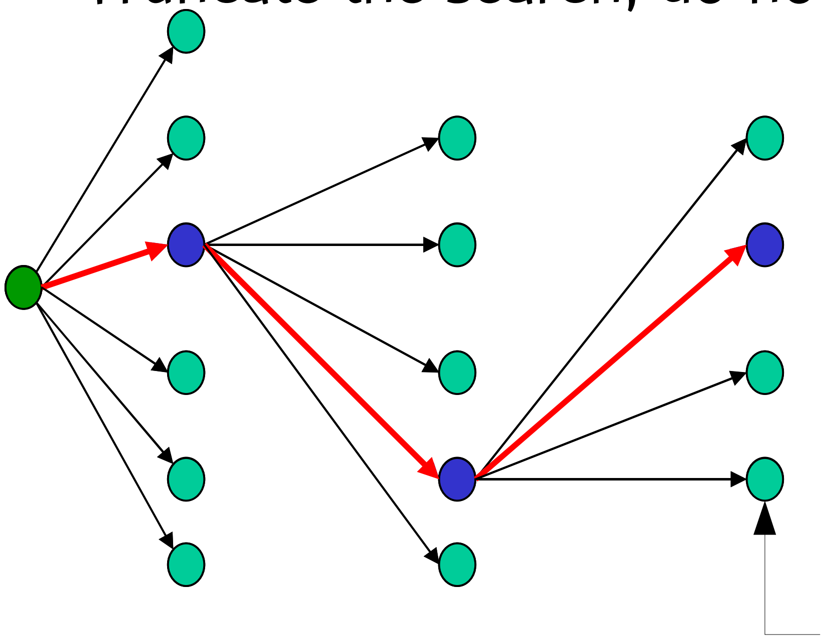
- Variants: trade-offs between computation time and solution quality
  - – Truncate the search, do not follow the full trajectory.



Truncate search here

Truncated path-relinking

- Variants: trade-offs between computation time and solution quality
  - Truncate the search, do not follow the full trajectory.



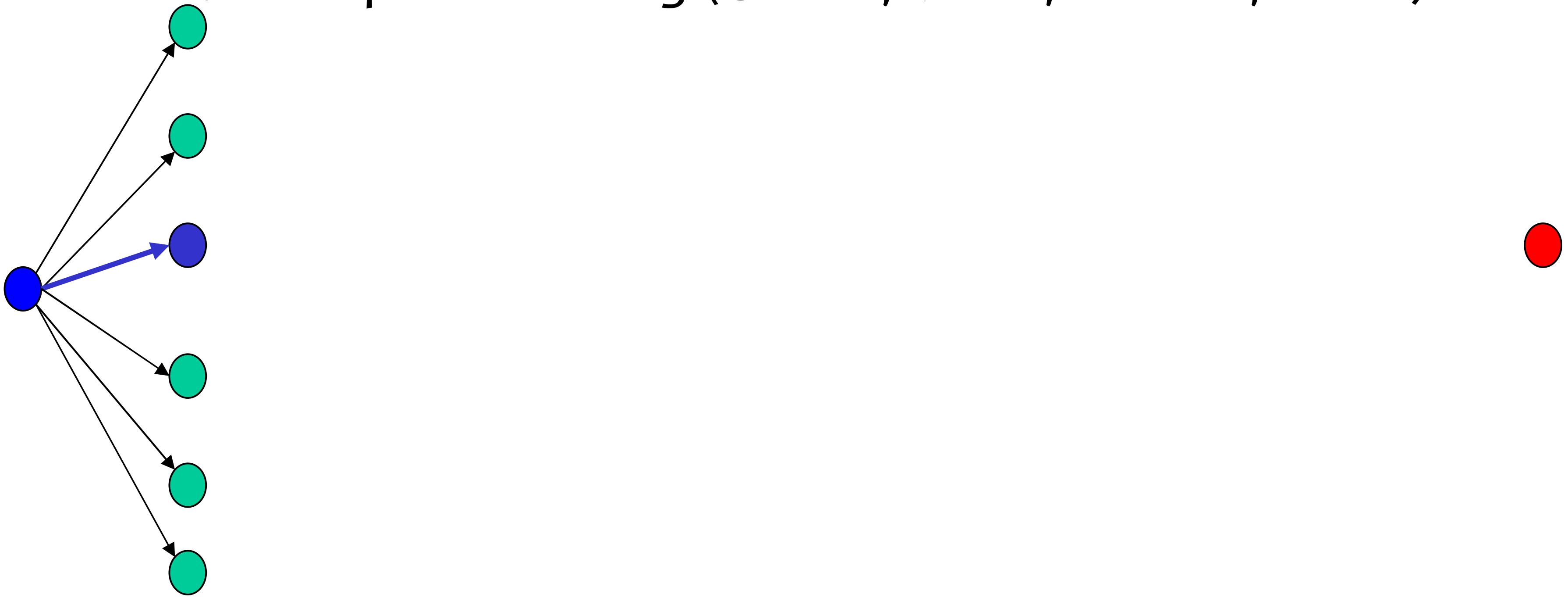
Truncate search here

Mixed path-relinking

- Variants: trade-offs between computation time and solution quality
  - Mixed path-relinking (Glover, 1997; Rosseti, 2003)

Mixed path-relinking

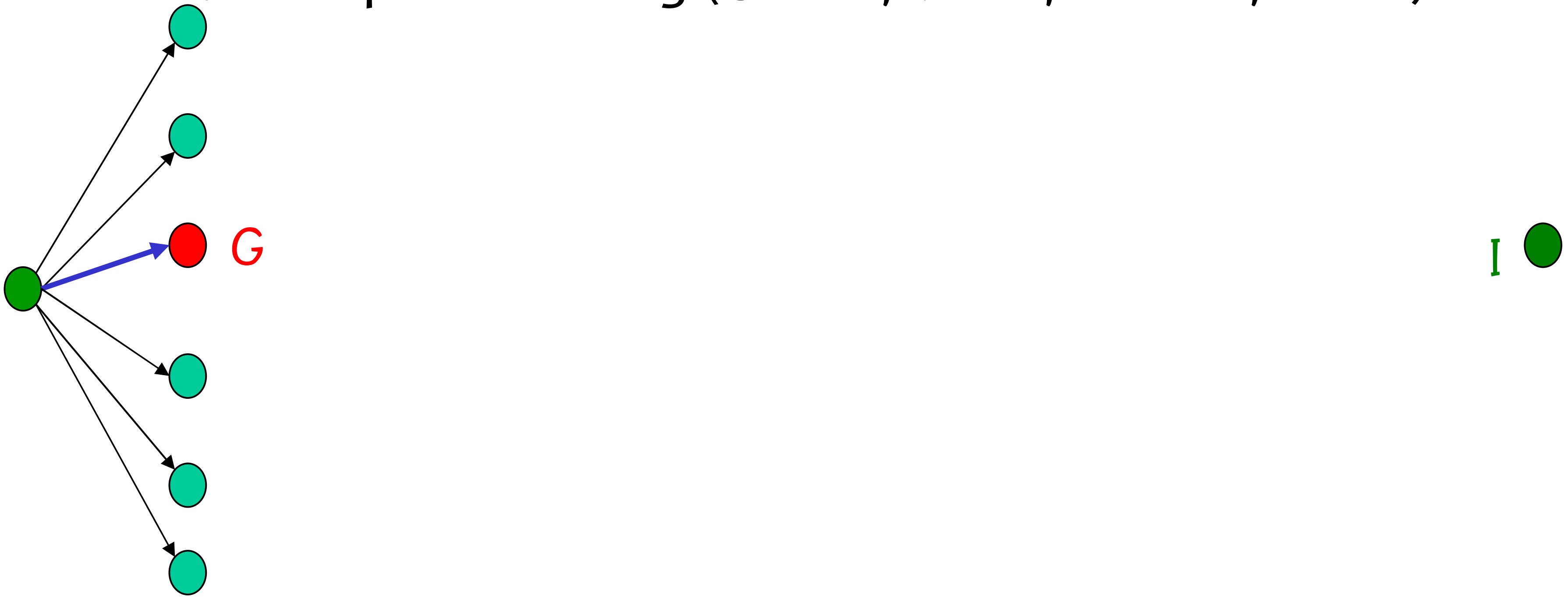
- Variants: trade-offs between computation time and solution quality
  - Mixed path-relinking (Glover, 1997; Rosseti, 2003)



Aug. 2007

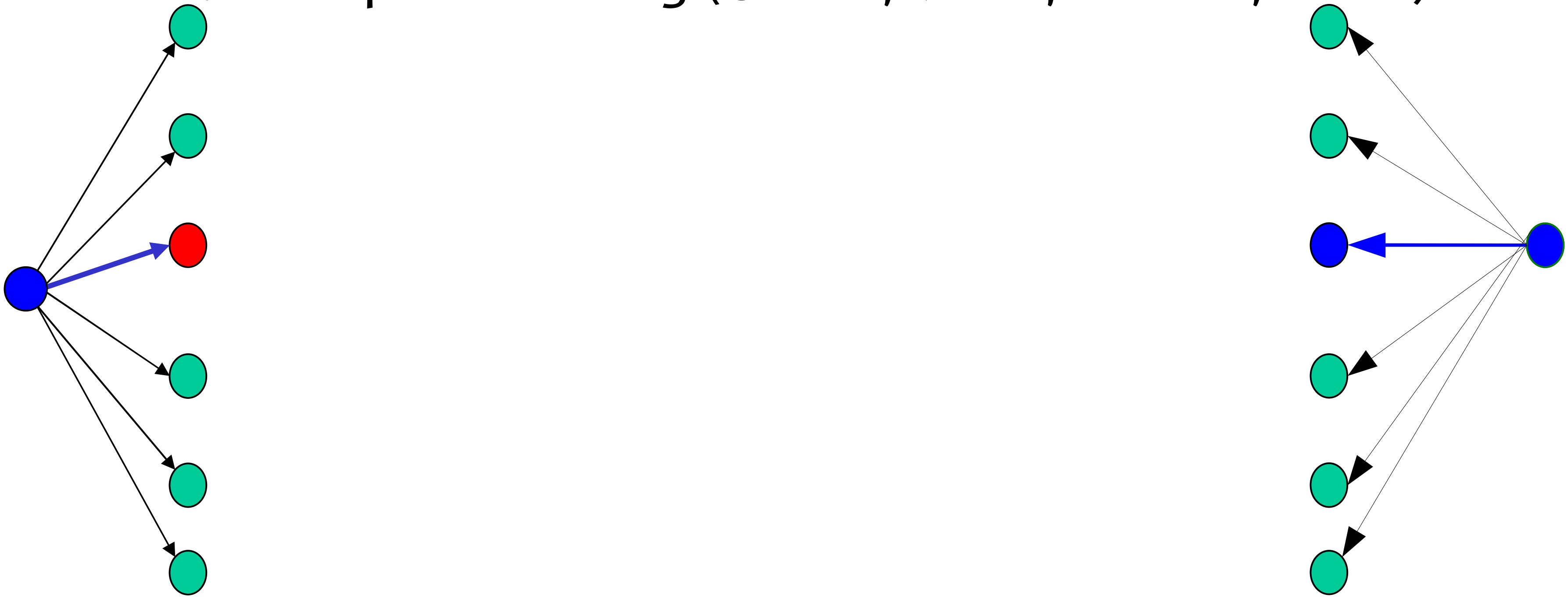
Mixed path-relinking

- Variants: trade-offs between computation time and solution quality
  - Mixed path-relinking (Glover, 1997; Rosseti, 2003)



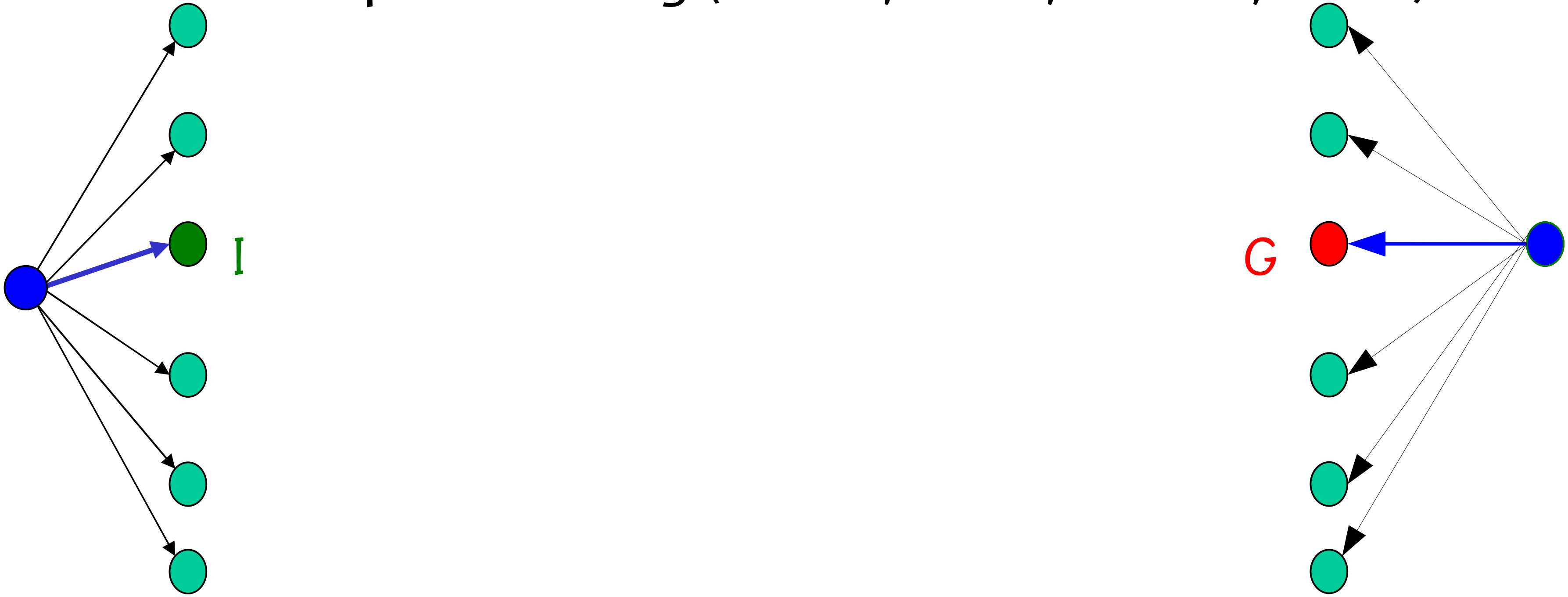
Mixed path-relinking

- Variants: trade-offs between computation time and solution quality
  - Mixed path-relinking (Glover, 1997; Rosseti, 2003)



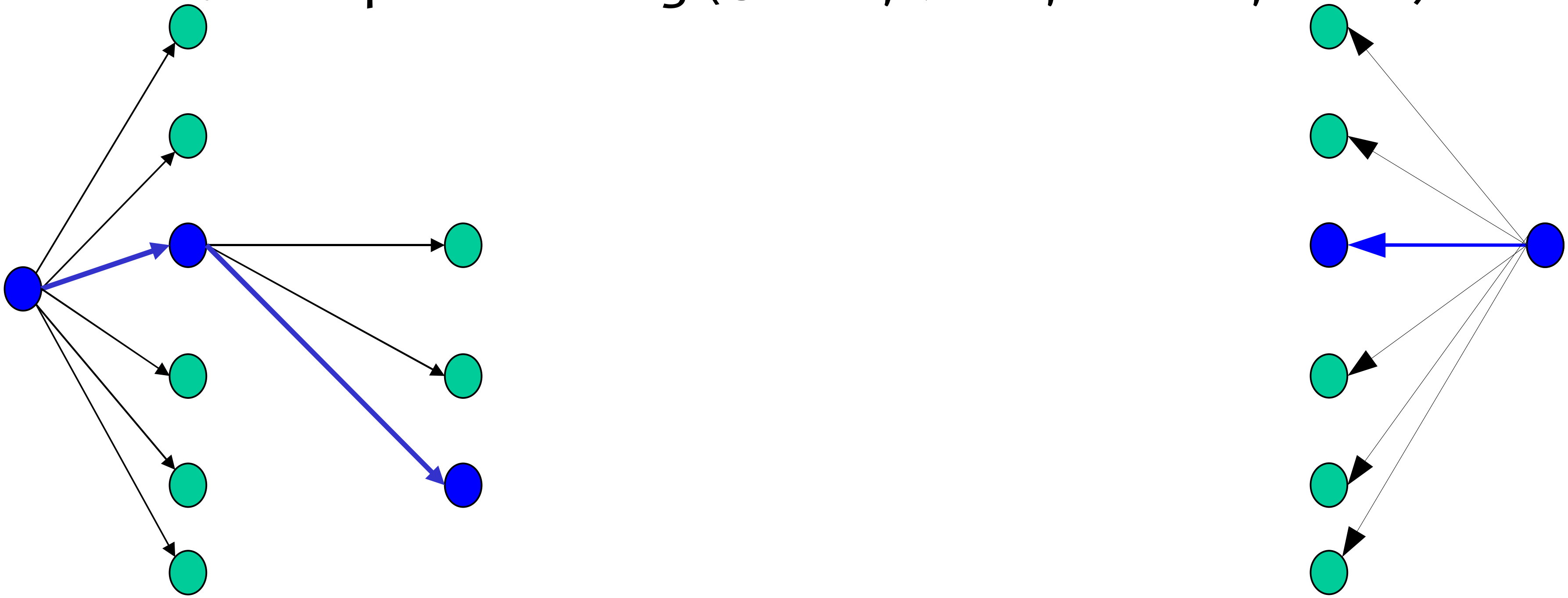
Mixed path-relinking

- Variants: trade-offs between computation time and solution quality
  - Mixed path-relinking (Glover, 1997; Rosseti, 2003)



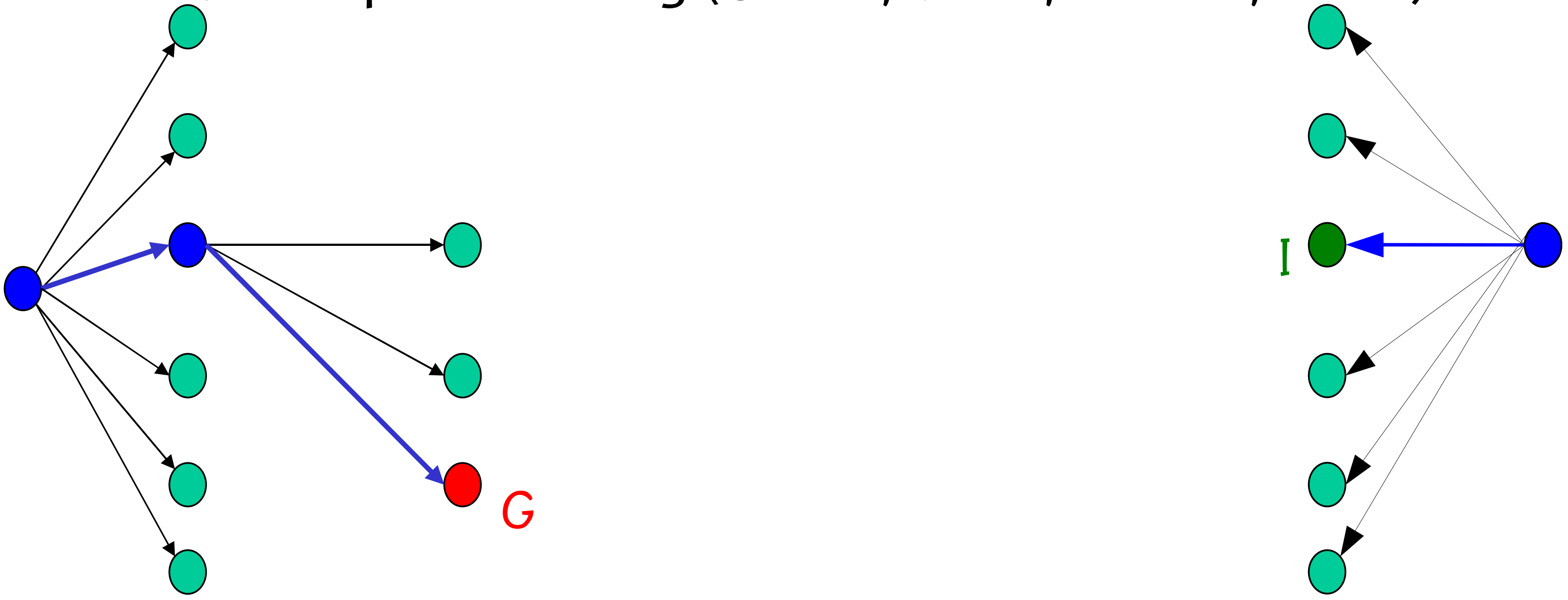
Mixed path-relinking

- Variants: trade-offs between computation time and solution quality
  - Mixed path-relinking (Glover, 1997; Rosseti, 2003)



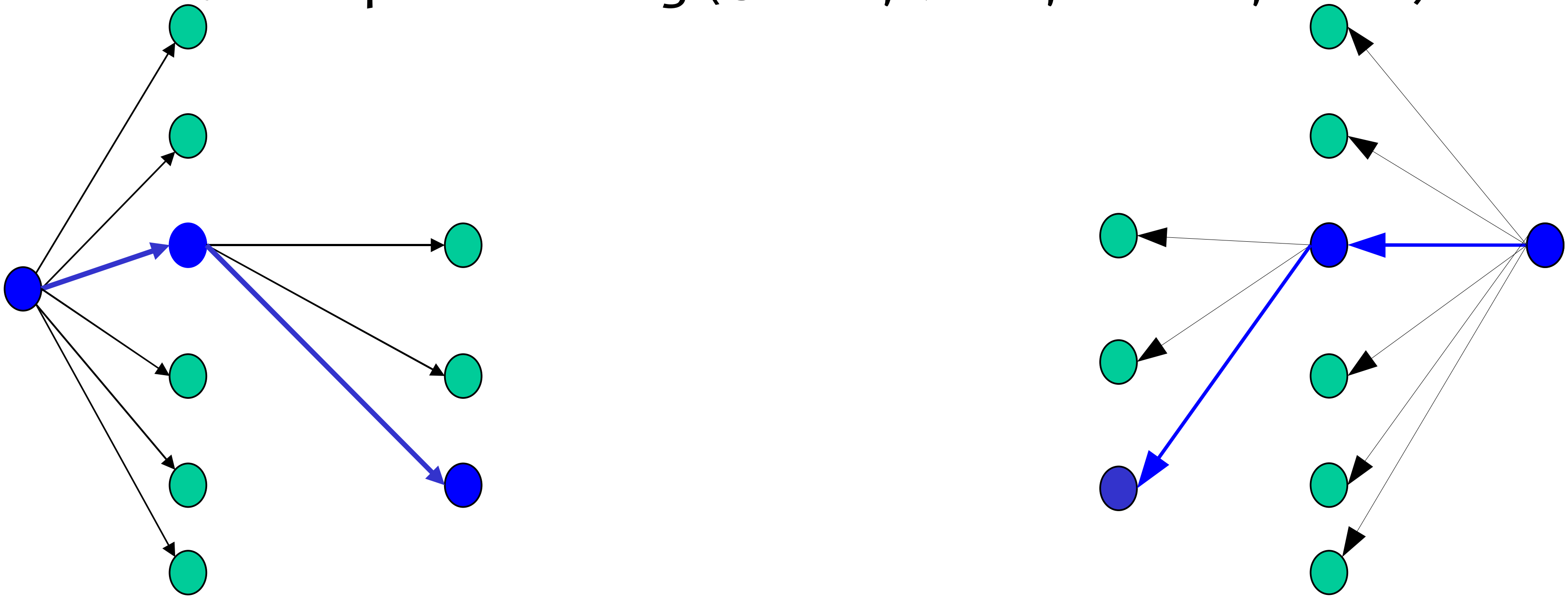
Mixed path-relinking

- Variants: trade-offs between computation time and solution quality
  - Mixed path-relinking (Glover, 1997; Rosseti, 2003)



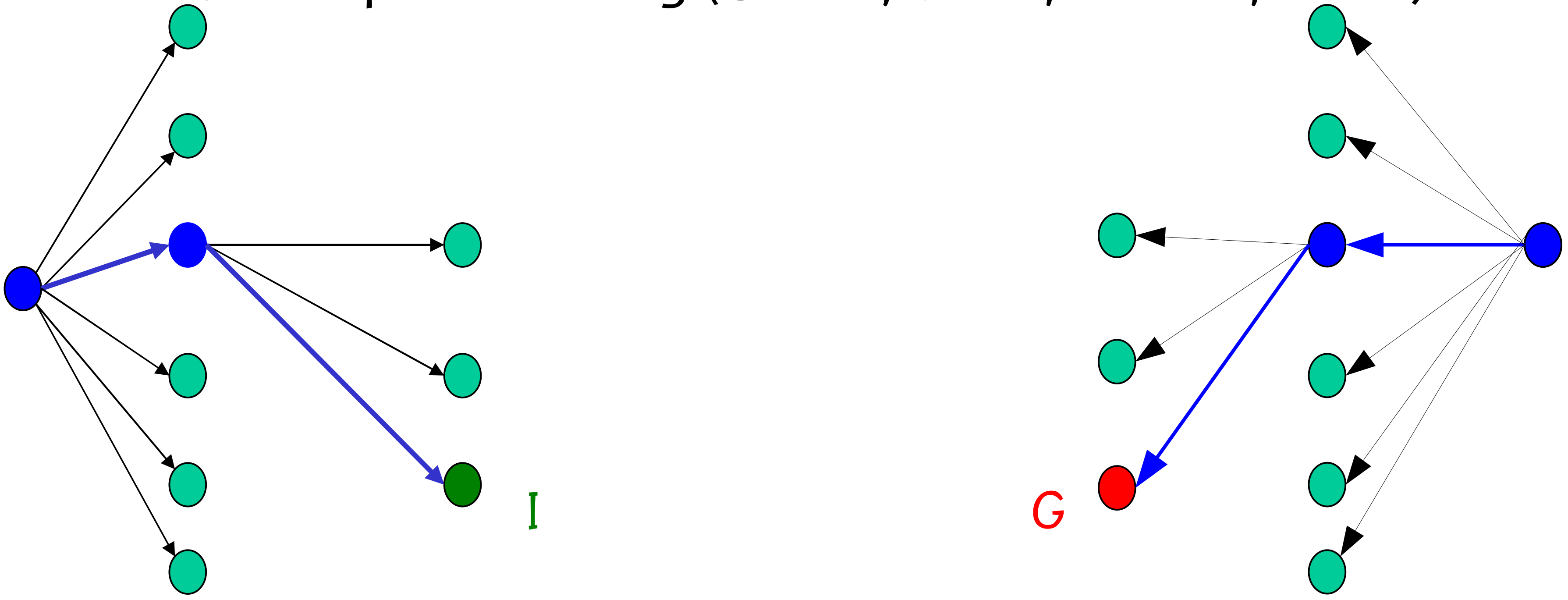
Mixed path-relinking

- Variants: trade-offs between computation time and solution quality
  - Mixed path-relinking (Glover, 1997; Rosseti, 2003)



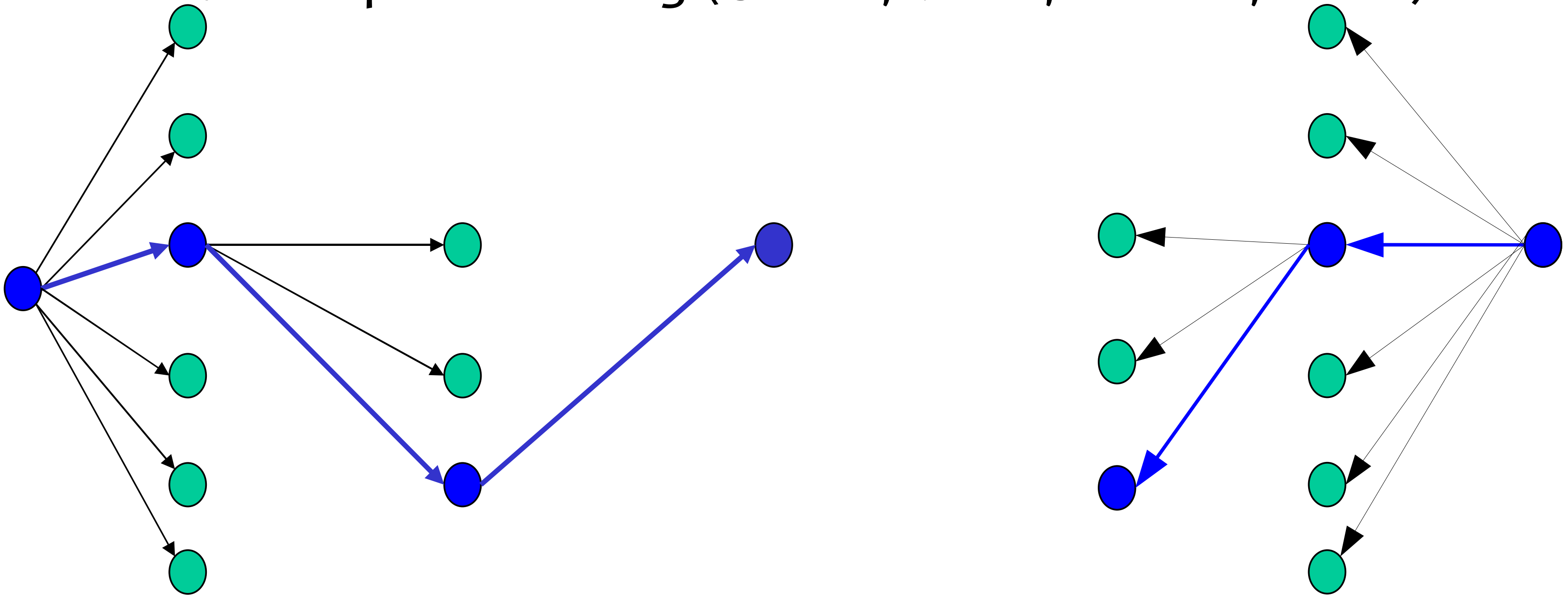
Mixed path-relinking

- Variants: trade-offs between computation time and solution quality
  - Mixed path-relinking (Glover, 1997; Rosseti, 2003)



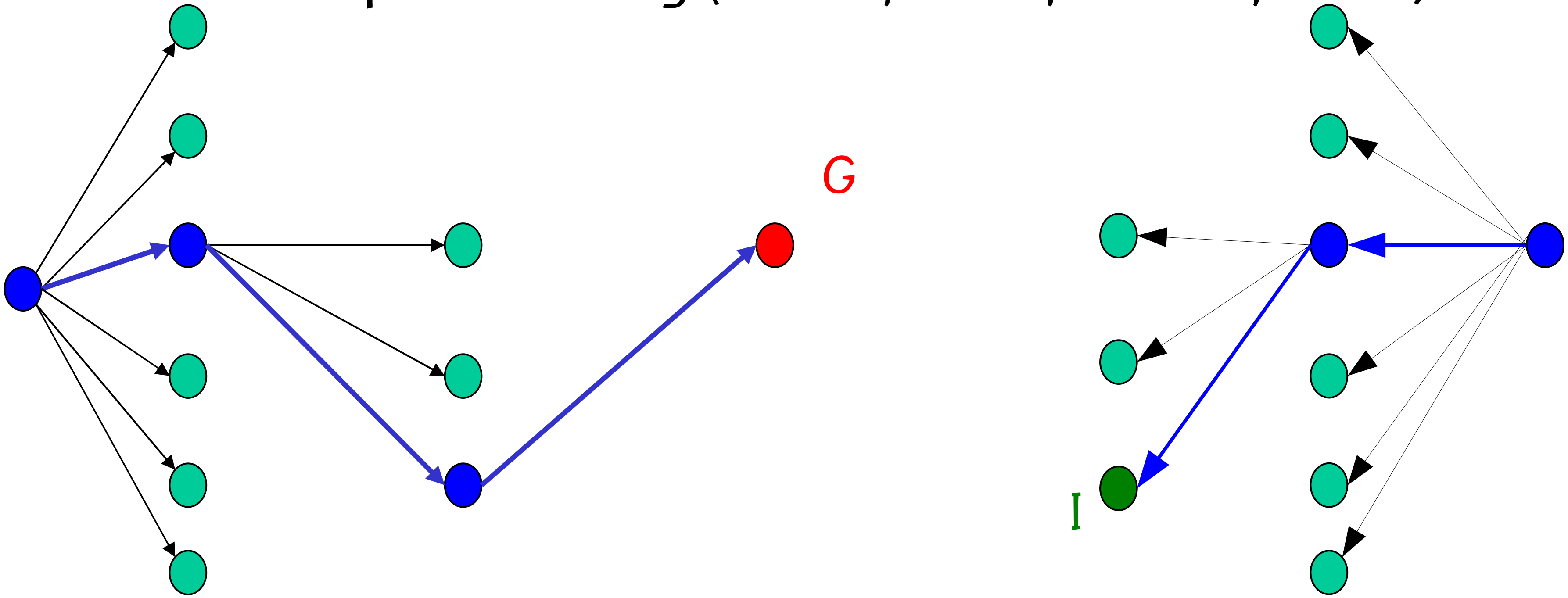
Mixed path-relinking

- Variants: trade-offs between computation time and solution quality
  - Mixed path-relinking (Glover, 1997; Rosseti, 2003)



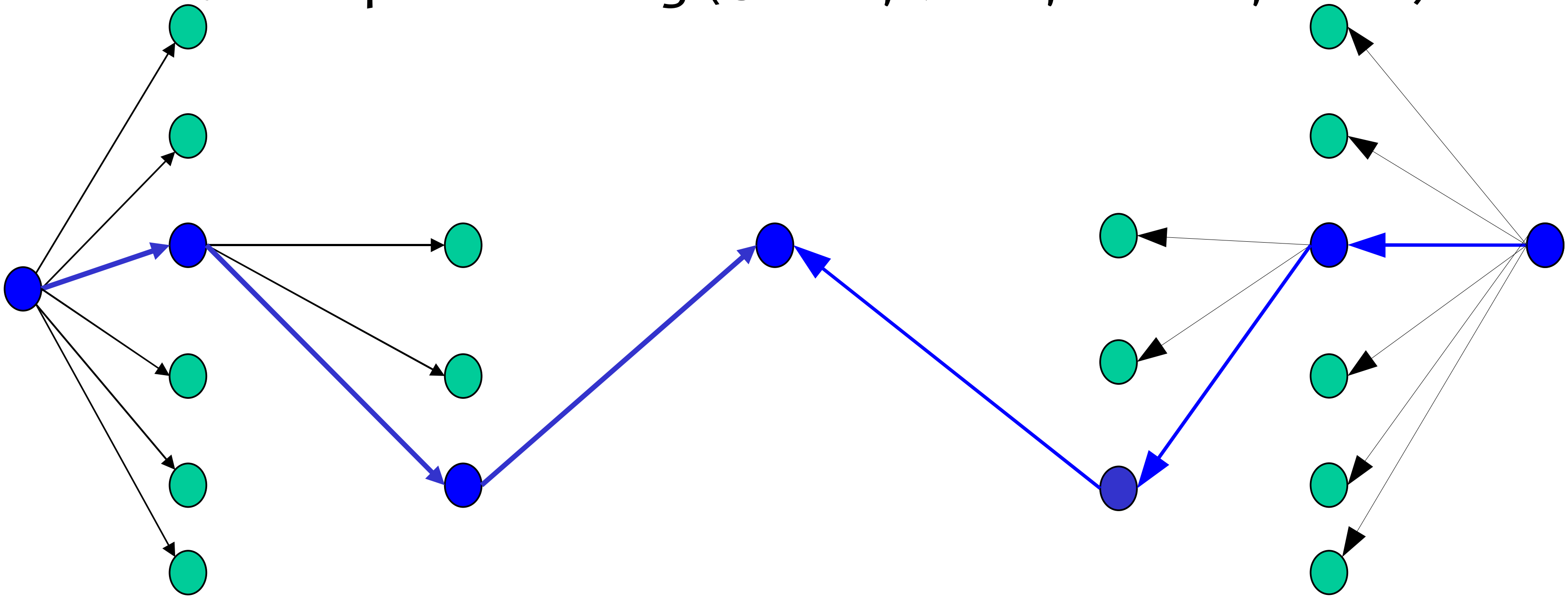
Mixed path-relinking

- Variants: trade-offs between computation time and solution quality
  - Mixed path-relinking (Glover, 1997; Rosseti, 2003)



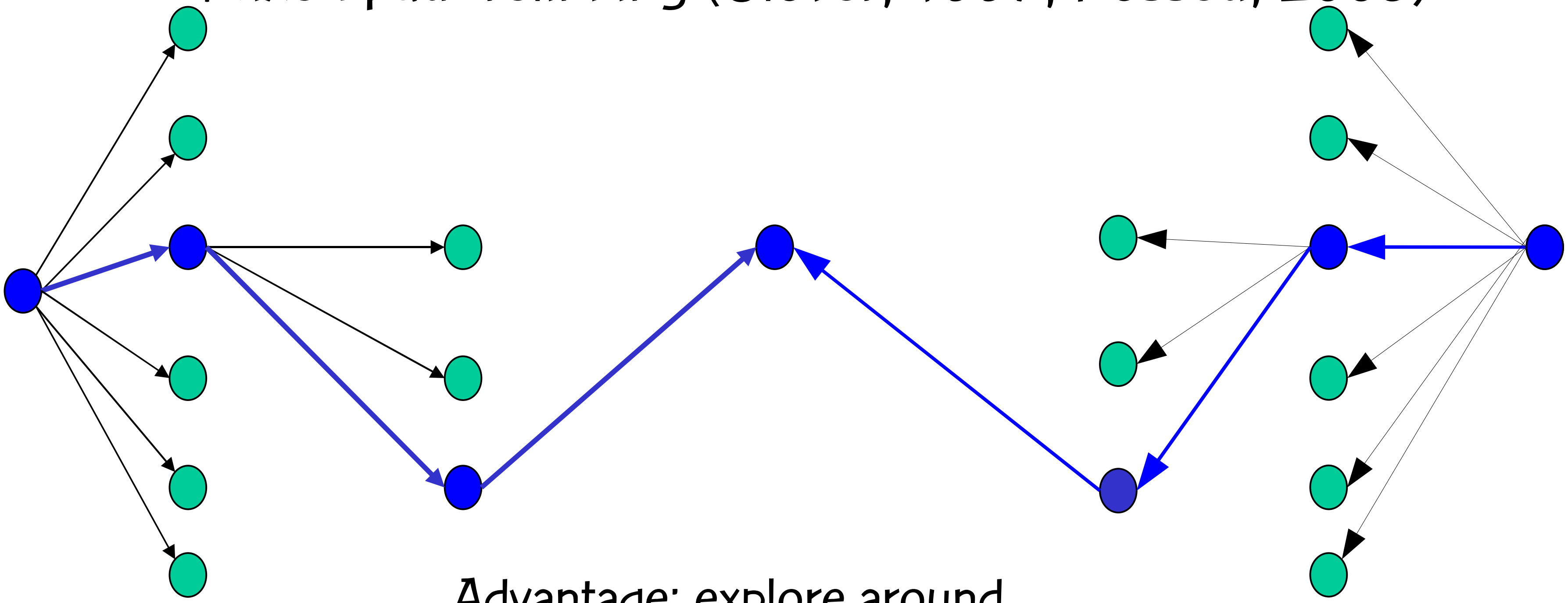
Mixed path-relinking

- Variants: trade-offs between computation time and solution quality
  - Mixed path-relinking (Glover, 1997; Rosseti, 2003)



Mixed path-relinking

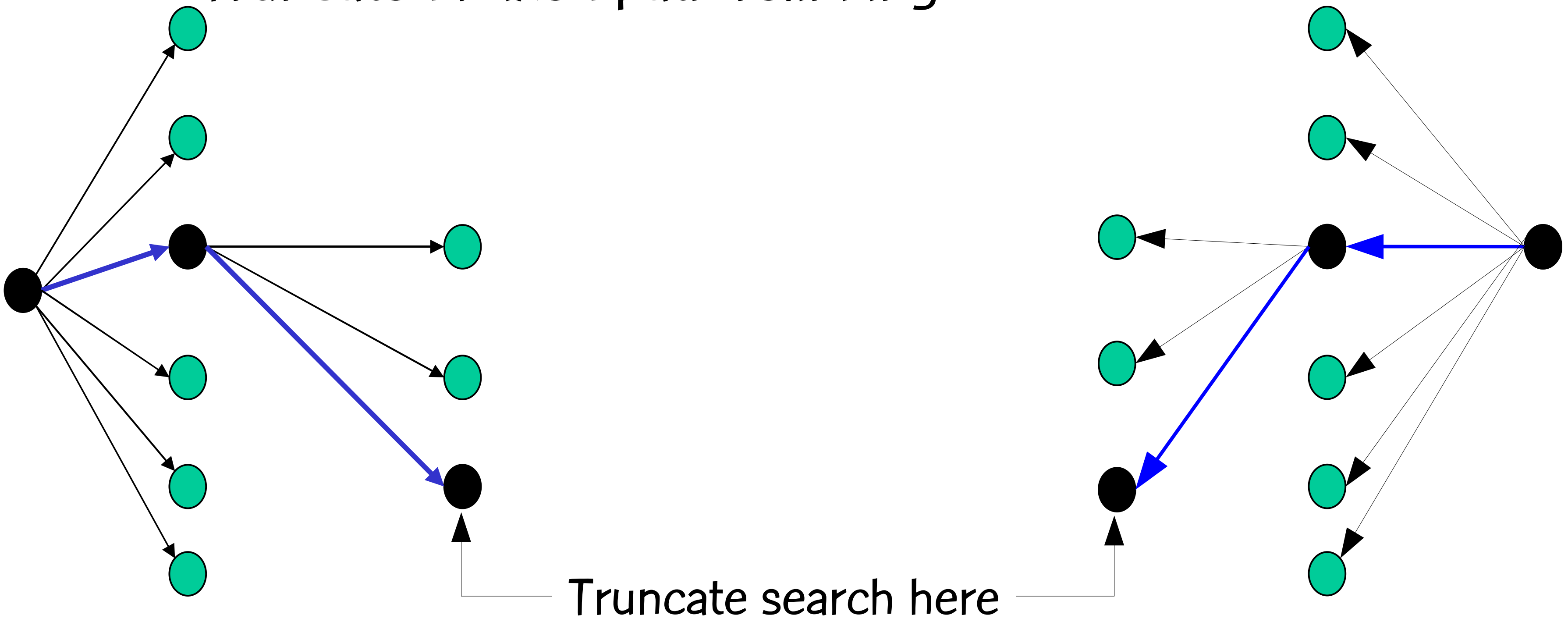
- Variants: trade-offs between computation time and solution quality
  - Mixed path-relinking (Glover, 1997; Rosseti, 2003)



Advantage: explore around neighborhoods of both input solutions.

Truncated mixed path-relinking

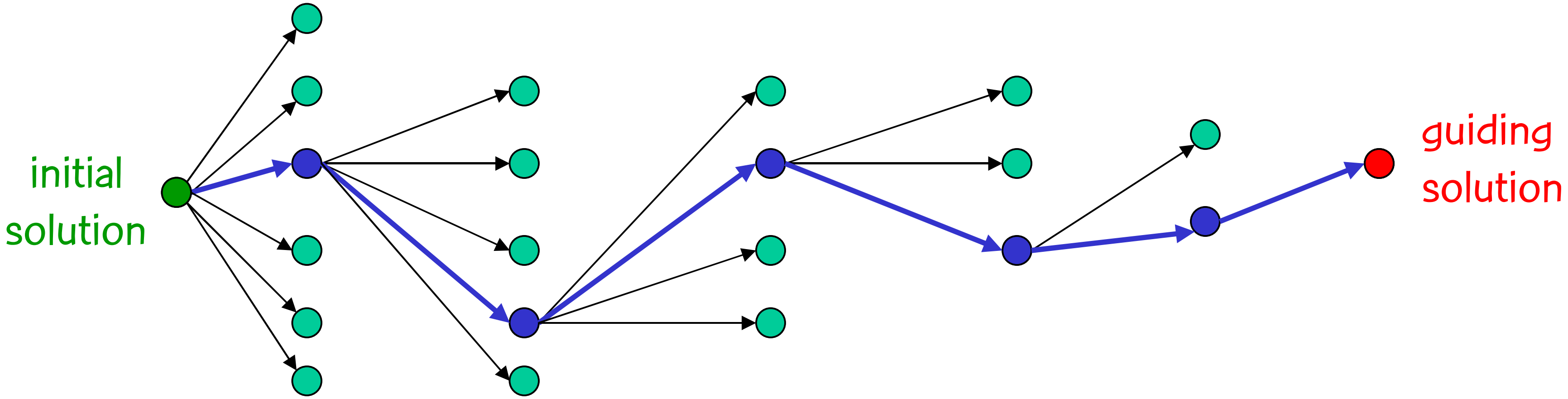
- Variants: trade-offs between computation time and solution quality
  - Truncated mixed path-relinking



Greedy randomized adaptive path-relinking

Faria, Binato, Resende, & Falcão (2001, 2005)

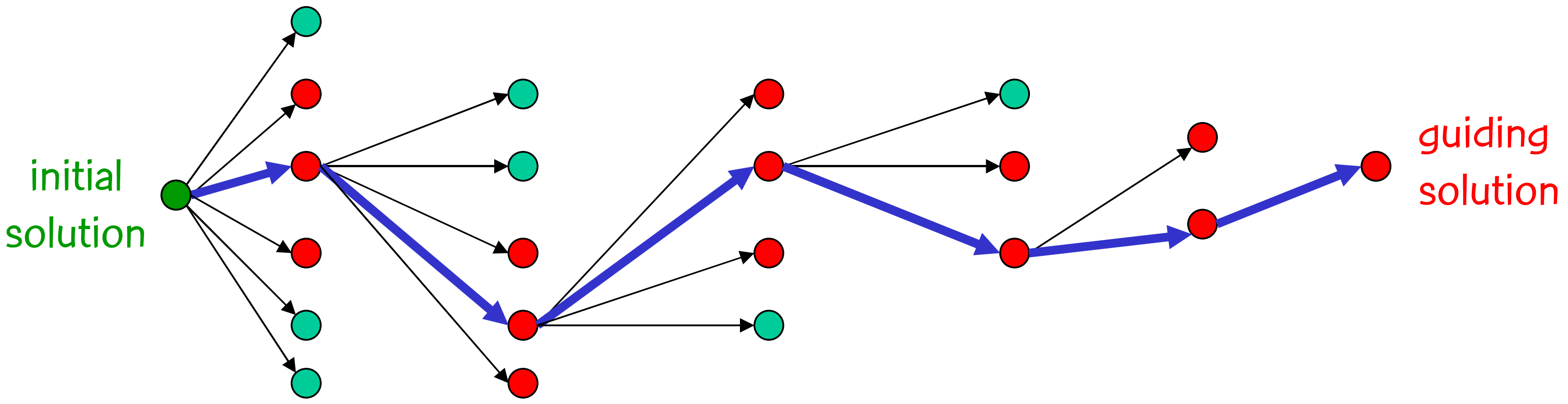
- Incorporates semi-greediness into PR.
- Standard PR selects moves greedily: samples one of exponentially many paths



Greedy randomized adaptive path-relinking

Faria, Binato, Resende, & Falcão (2001, 2005)

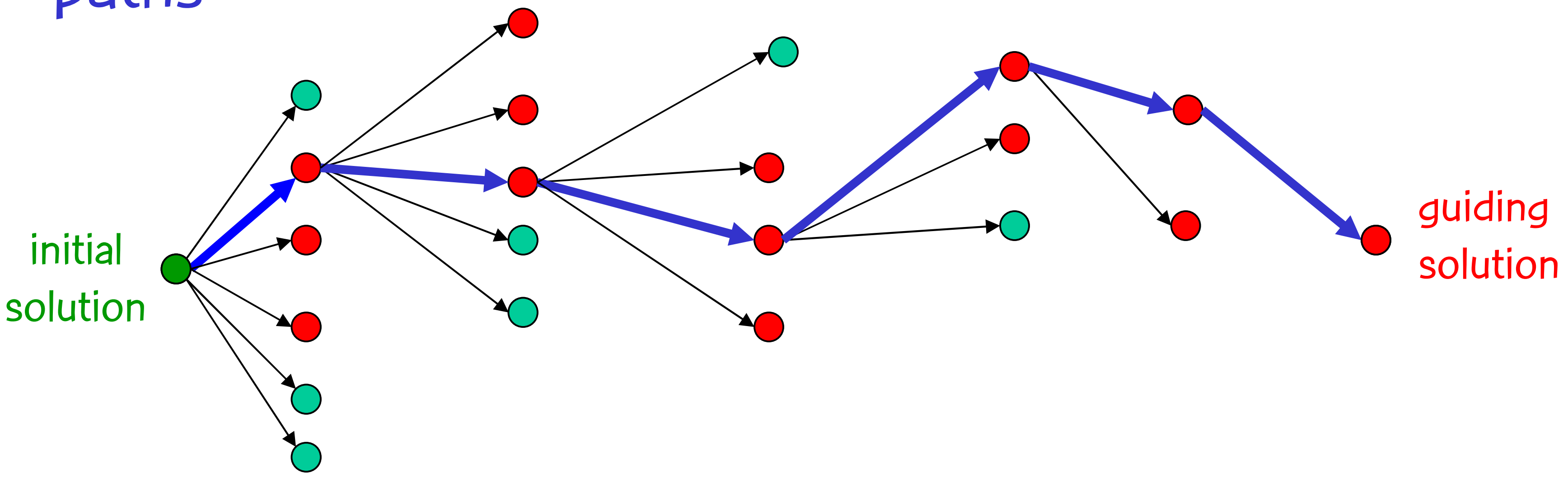
- Incorporates semi-greediness into PR.
- graPR creates RCL with best moves: samples several paths



Greedy randomized adaptive path-relinking

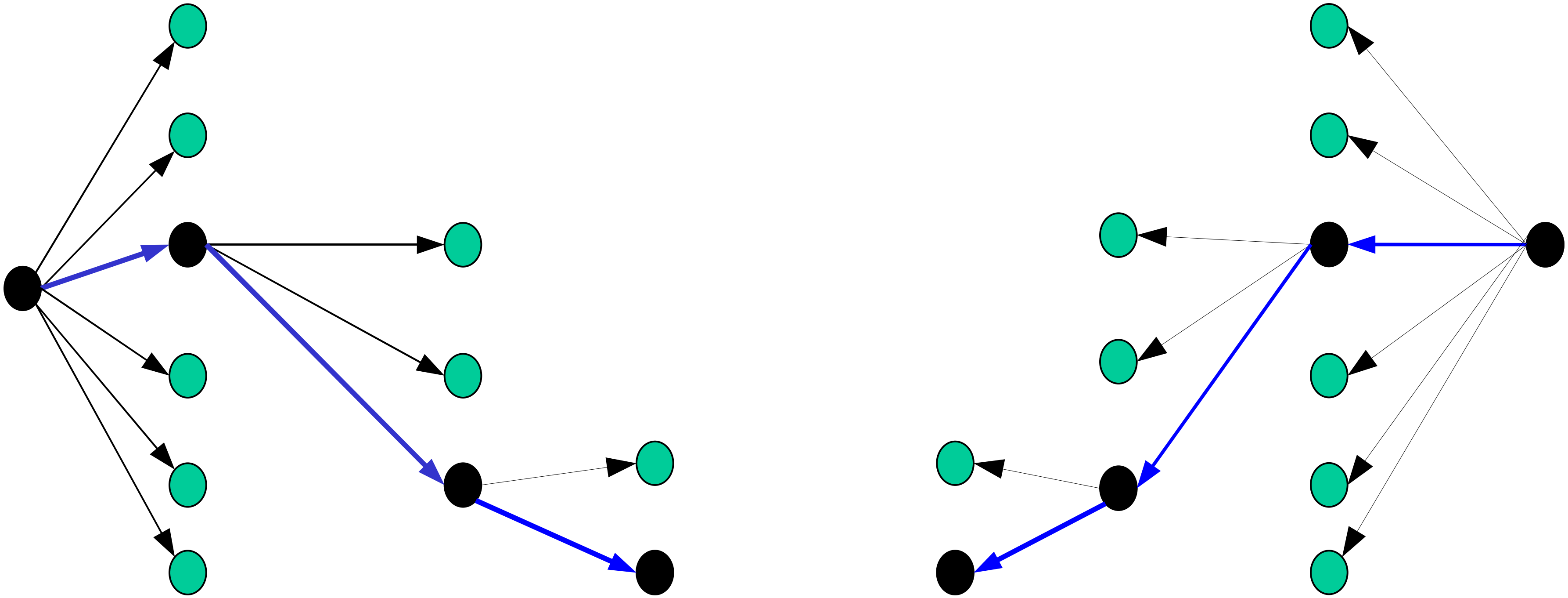
Faria, Binato, Resende, & Falcão (2001, 2005)

- Incorporates semi-greediness into PR.
- graPR creates RCL with best moves: samples several paths



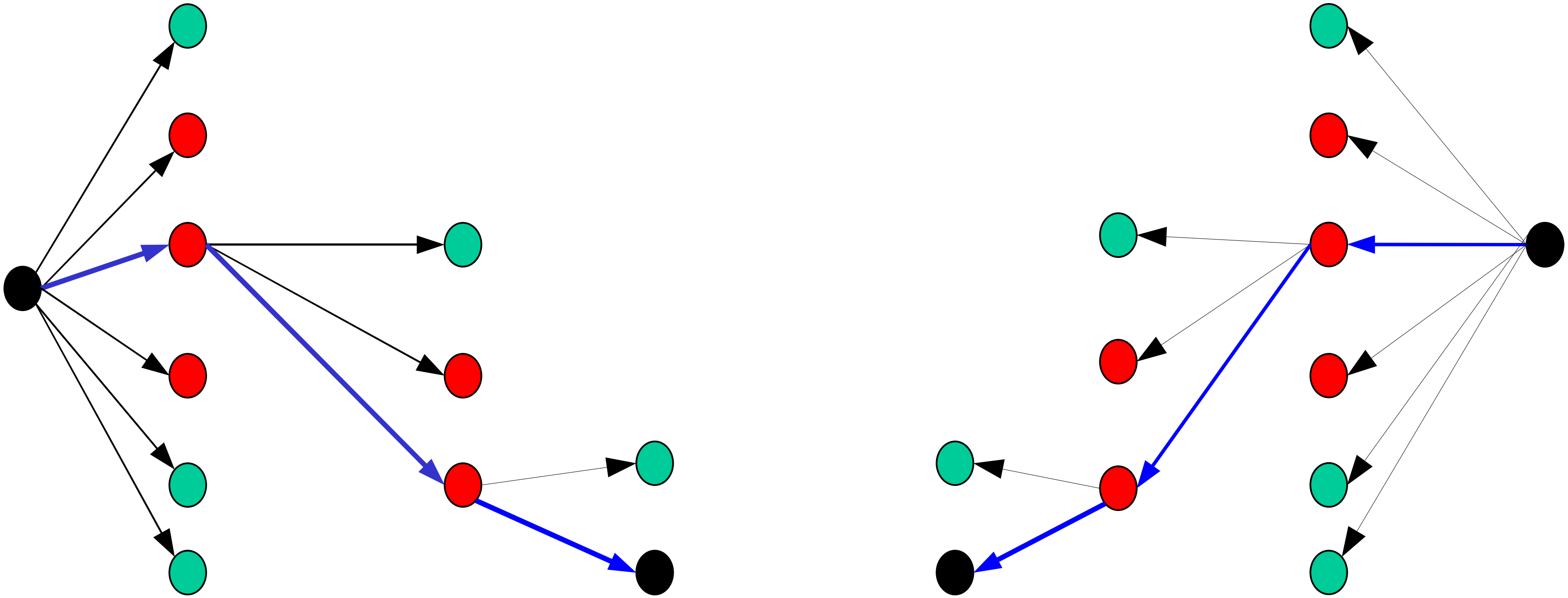
Truncated mixed graPR

When applied to a given pair of solutions truncated mixed PR explores one of exponentially many path segments each time it is executed.



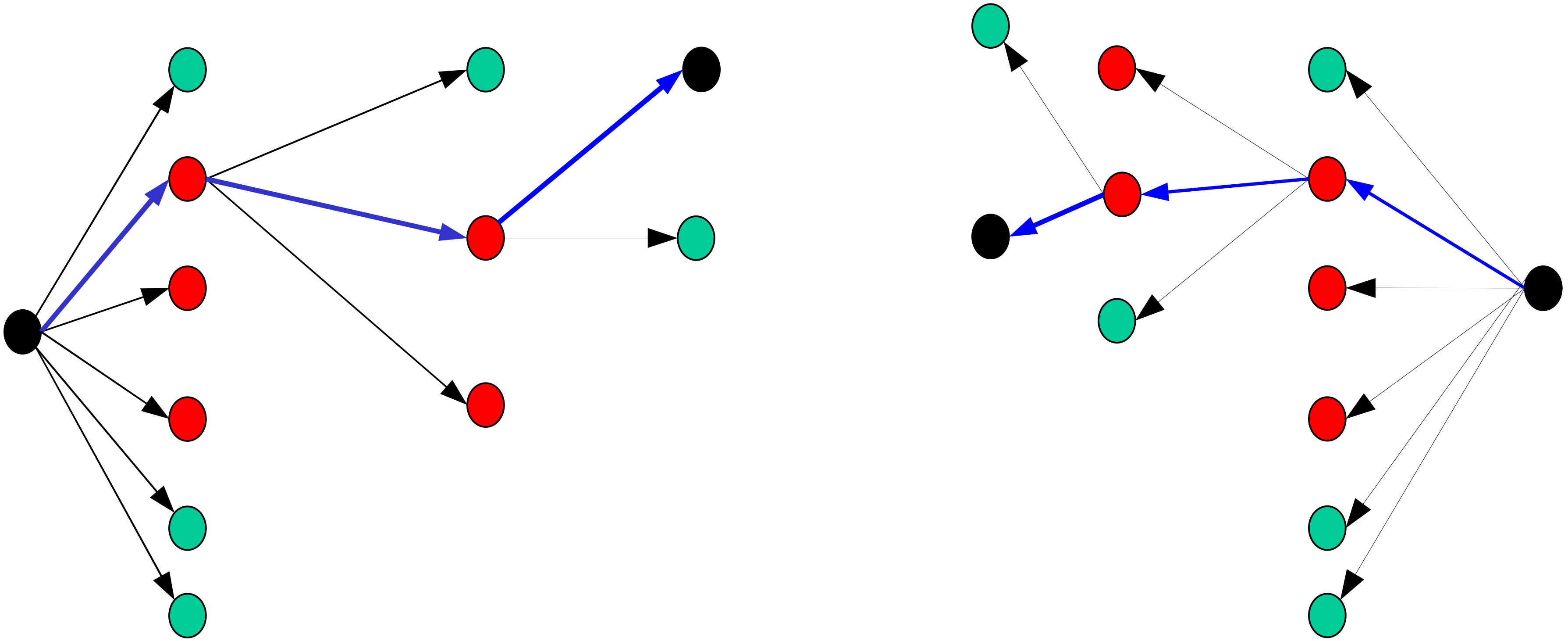
Truncated mixed graPR

With high probability, truncated mixed graPR explores different path segments each time it is executed between the same pair of solutions.



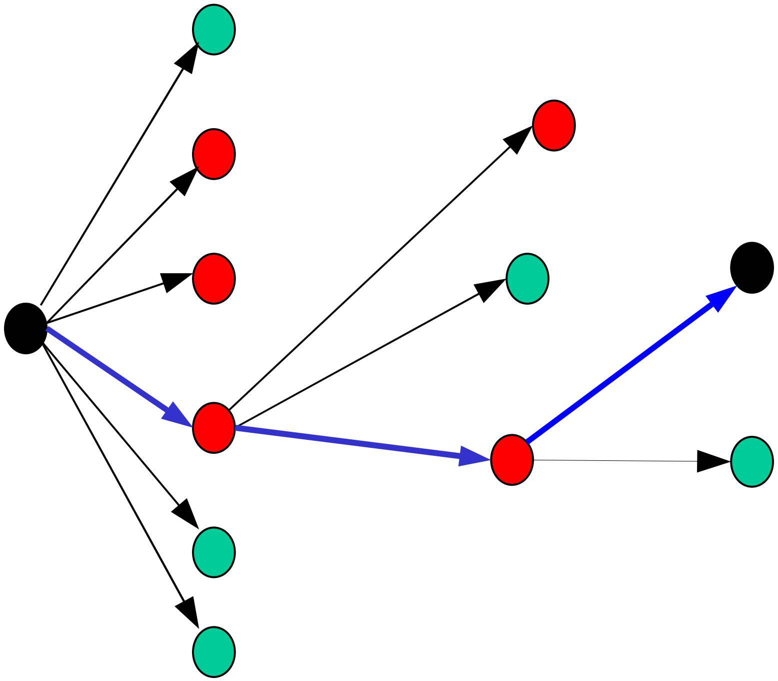
Truncated mixed graPR

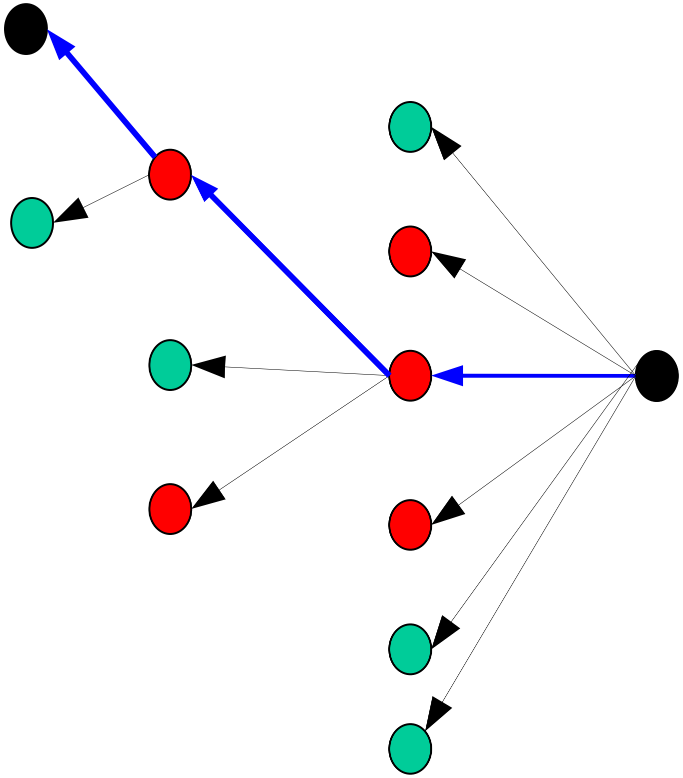
With high probability, truncated mixed graPR explores different path segments each time it is executed between the same pair of solutions.



Truncated mixed graPR

With high probability, truncated mixed graPR explores different path segments each time it is executed between the same pair of solutions.





GRASP with path-relinking

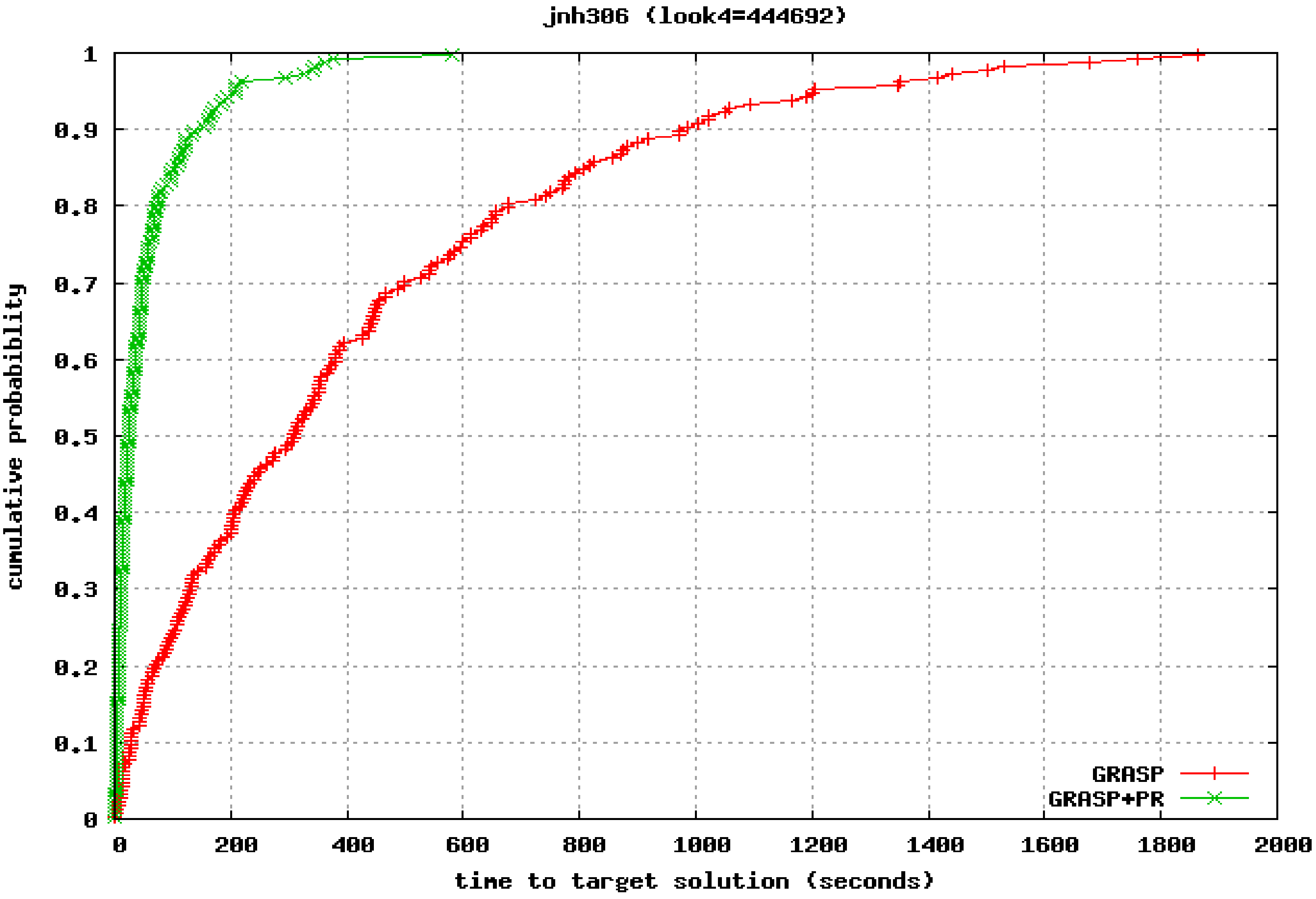
GRASP with path-relinking

- First proposed by Laguna and Martí (1999).
- Maintains a set of elite solutions found during GRASP iterations.
- After each GRASP iteration (construction and local search):
  - Use GRASP solution as initial solution.
  - Select an elite solution uniformly at random: guiding solution.
  - Perform path-relinking between these two solutions.

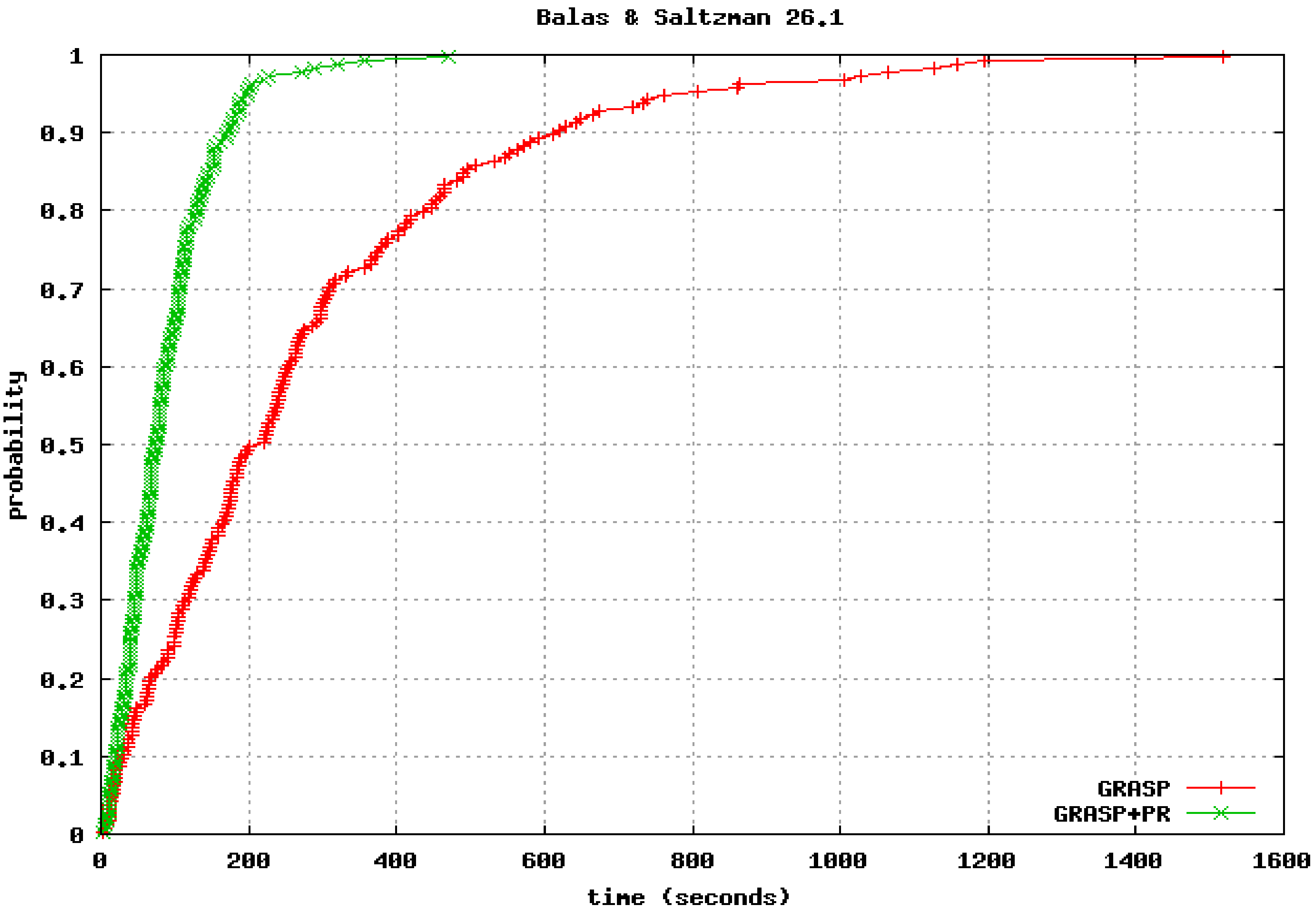
GRASP with path-relinking

- Since 1999, there has been a lot of activity in hybridizing GRASP with path-relinking.
- Survey by Resende & Ribeiro in MIC 2003 book of Ibaraki, Nonobe, and Yagiura (2005).
- Main observation from experimental studies: GRASP with path-relinking outperforms pure GRASP.

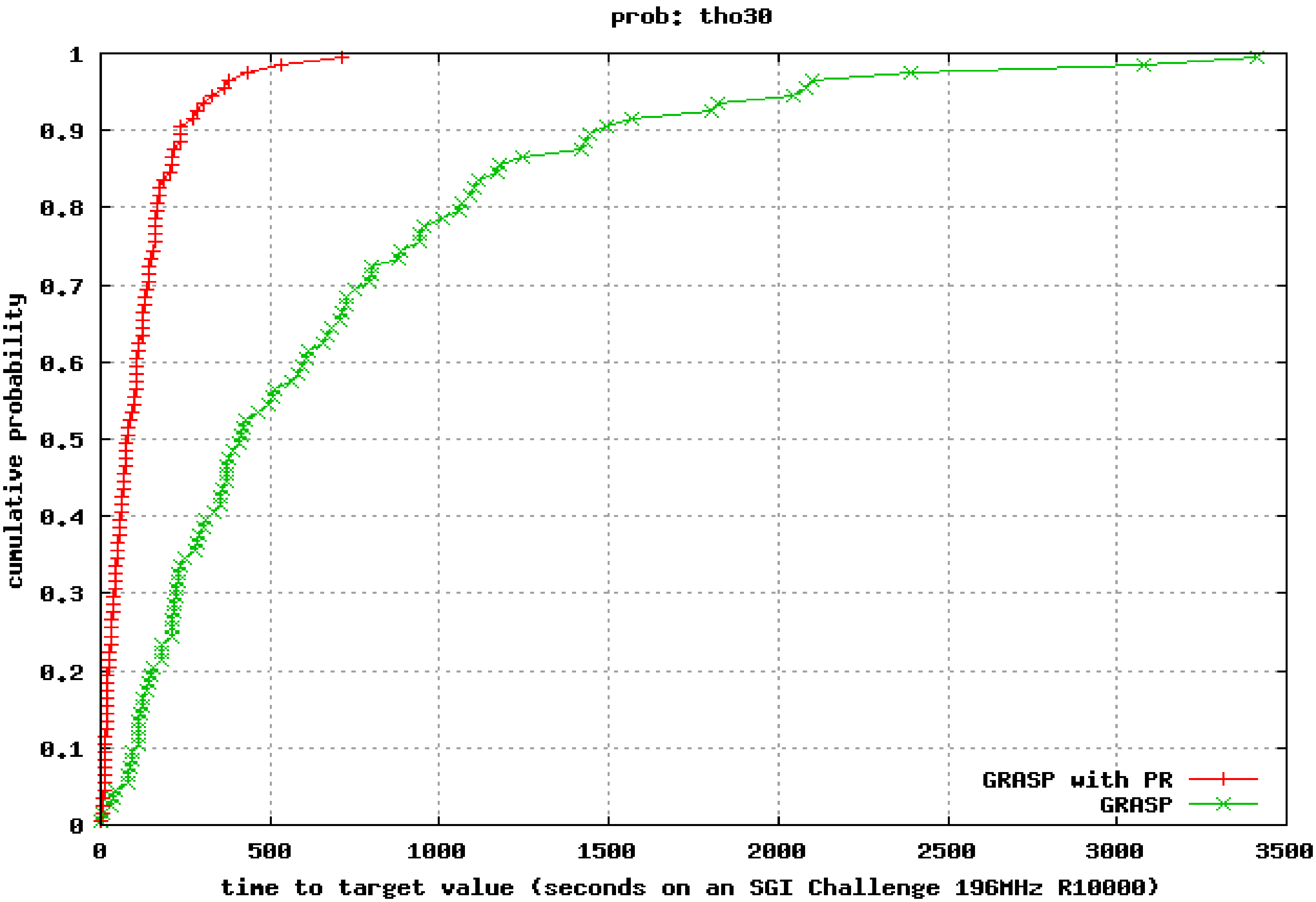
MAX-SAT (Festa, Pardalos, Pitsoulis, and Resende, 2006)



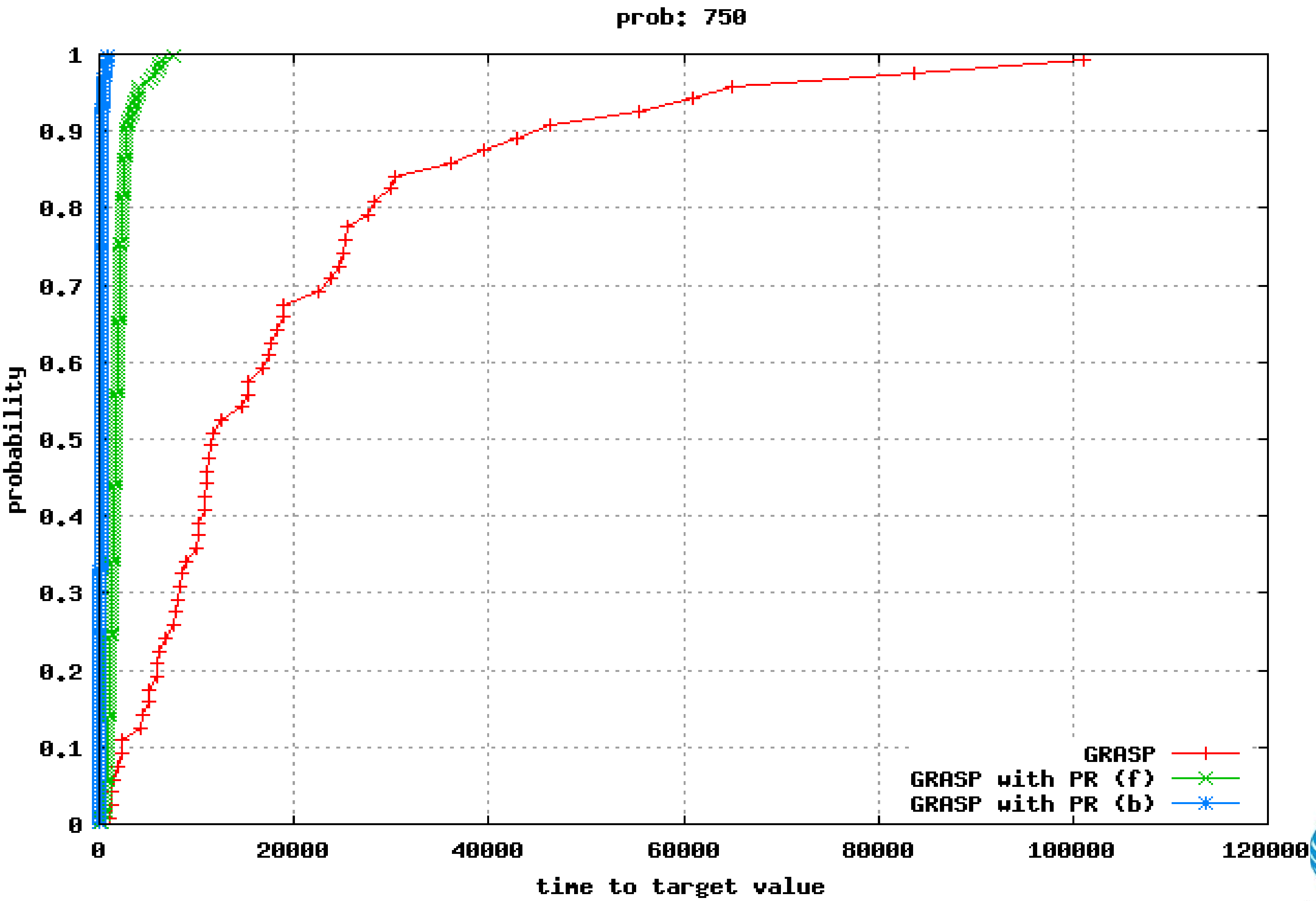
3-index assignment (Aiex, Resende, Pardalos, & Toraldo, 2005)



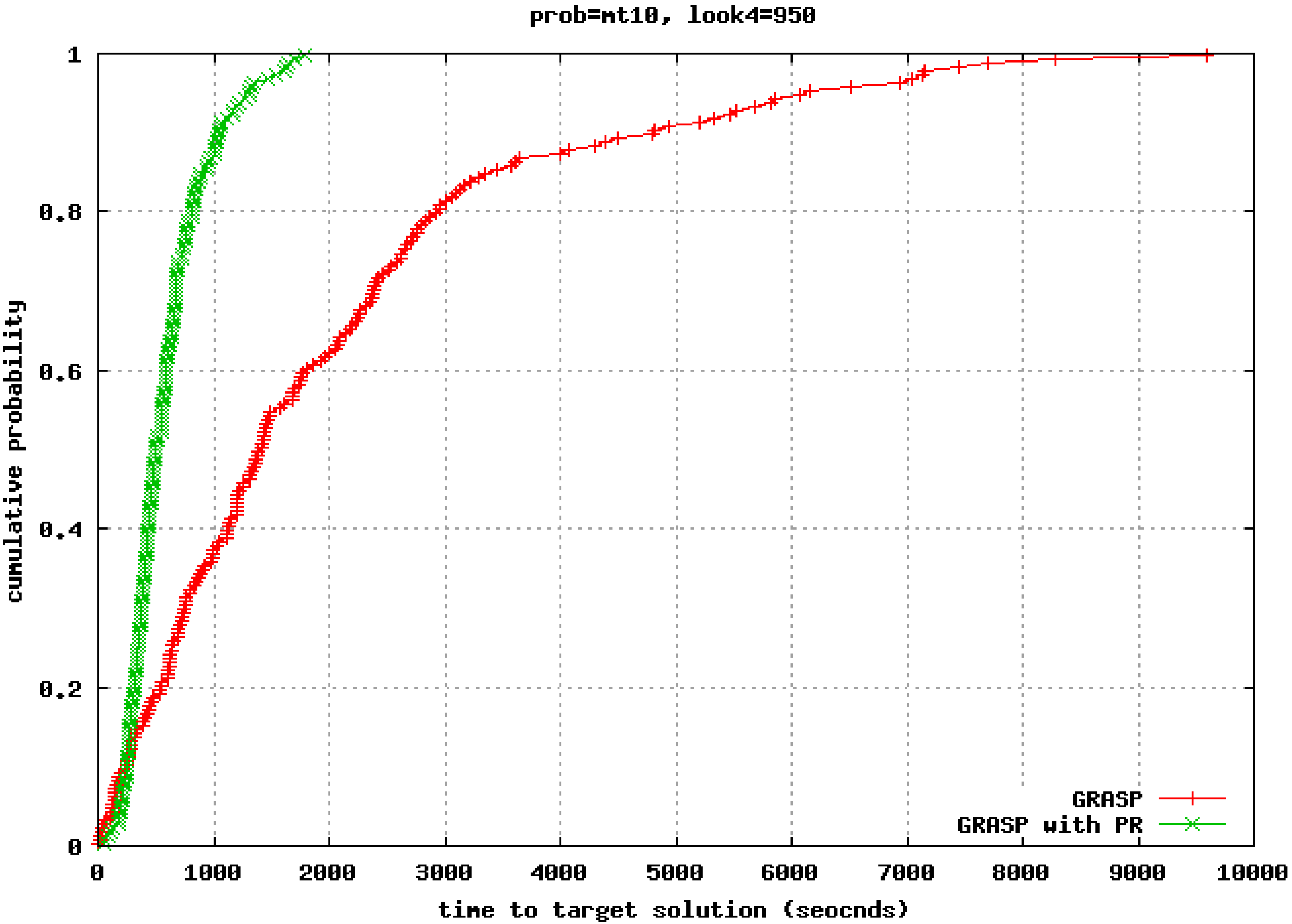
QAP (Oliveira, Pardalos, and Resende, 2004)



Bandwidth packing (Resende and Ribeiro, 2003)



Job shop scheduling (Aiex, Binato, & Resende, 2003)



GRASP with path-relinking:
Pool management

- P is a set (pool) of elite solutions.
- Ideally, pool has a set of good diverse solutions.
- Mechanisms are needed to guarantee that pool is made up of those kinds of solutions.

GRASP with path-relinking:
Pool management

- Each iteration of first |P| GRASP iterations adds one solution to P (if different from others).
- After that: solution x is promoted to P if:
  - x is better than best solution in P.
  - x is not better than best solution in P, but is better than worst and is sufficiently different from all solutions in P.

GRASP with path-relinking:

Pool management

- GRASP with PR works best when paths in PR are long, i.e. when the symmetric difference between the initial and guiding solutions is large.
- Given a solution to relink with an elite solution, which elite solution to choose?
  - Choose at random with probability proportional to the symmetric difference.

GRASP with path-relinking:

Pool management

- Solution quality and diversity are two goals of pool design.
- Given a solution X to insert into the pool, which elite solution do we choose to remove?
  - Of all solutions in the pool with worse solution than X, select to remove the pool solution most similar to X, i.e. with the smallest symmetric difference from X.

GRASP with path-relinking

Repeat GRASP with PR loop

1) Construct randomized greedy X
2) Y = local search to improve X
3) Path-relinking between Y and pool solution Z
4) Update pool

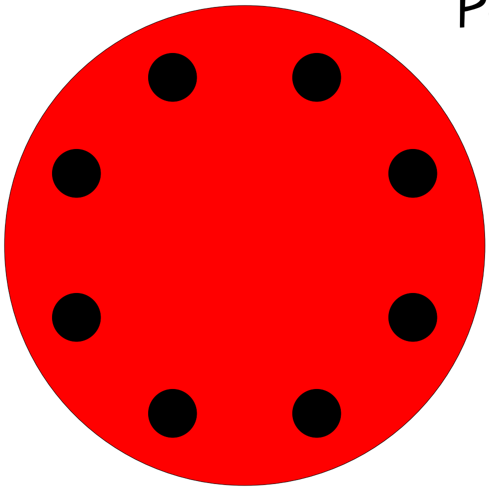
Evolutionary path-relinking (EvPR)

Evolutionary path-relinking

(Resende & Werneck, 2004, 2006)

- Evolutionary path-relinking “evolves” the pool, i.e. transforms it into a pool of diverse elements whose solution values are better than those of the original pool.
- Evolutionary path-relinking can be used
  - as an intensification procedure at certain points of the solution process;
  - as a post-optimization procedure at the end of the solution process.

Evolutionary path-relinking (EvPR)

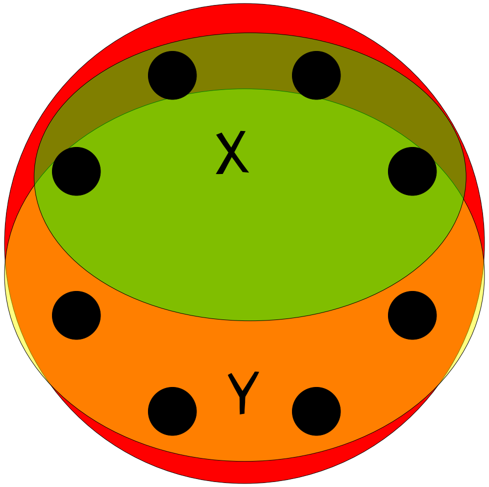


Population P(0)

Each “population” of EvPR starts with a pool of elite solutions of size |P|.

Population P(0) is the current elite set.

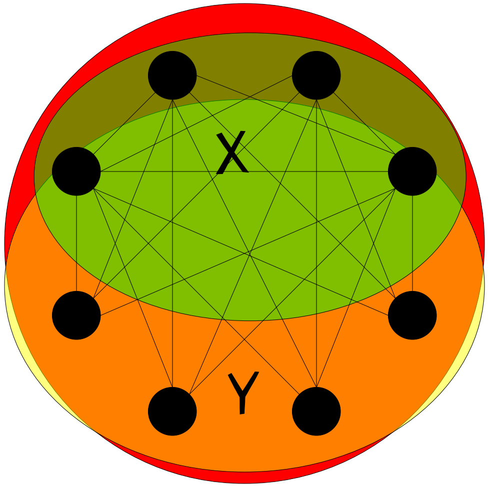
Evolutionary path-relinking (EvPR)



All pairs of elite solutions (x,y) in K-th population P(K), such that $x \in X \subseteq P(K)$ and $y \in Y \subseteq P(K)$, are path-relinked and the resulting $z = PR(x,y)$ is a candidate for inclusion in population P(K+1).

Rules for inclusion into P(K+1) are the same used for inclusion into any pool.

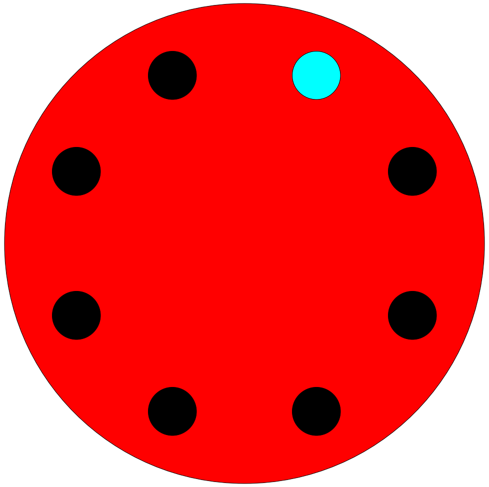
Evolutionary path-relinking (EvPR)



All pairs of elite solutions (x,y) in K-th population P(K), such that $x \in X \subseteq P(K)$ and $y \in Y \subseteq P(K)$, are path-relinked and the resulting $z = PR(x,y)$ is a candidate for inclusion in population P(K+1).

Rules for inclusion into P(K+1) are the same used for inclusion into any pool.

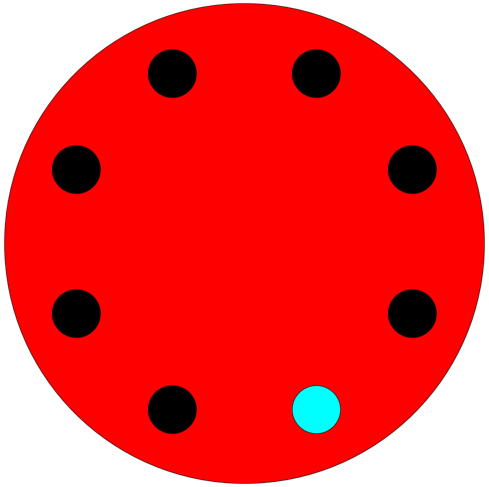
Evolutionary path-relinking (EvPR)



Population P(K)

If best solution in population P(K+1) has same objective function value as best solution in population P(K), process stops.

Else K=K+1 and repeat.

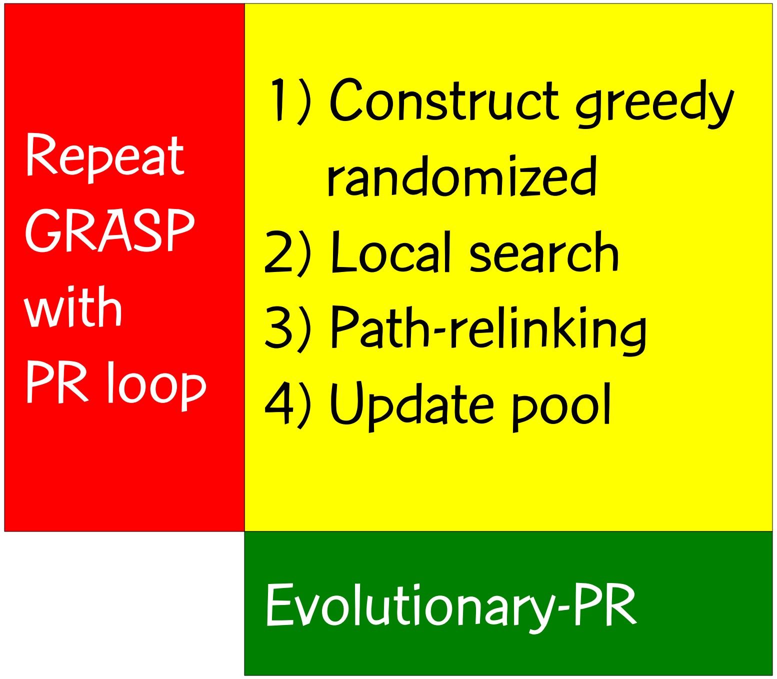


Population P(K+1)

GRASP with evolutionary path-relinking

GRASP with evolutionary path-relinking

As post-optimization

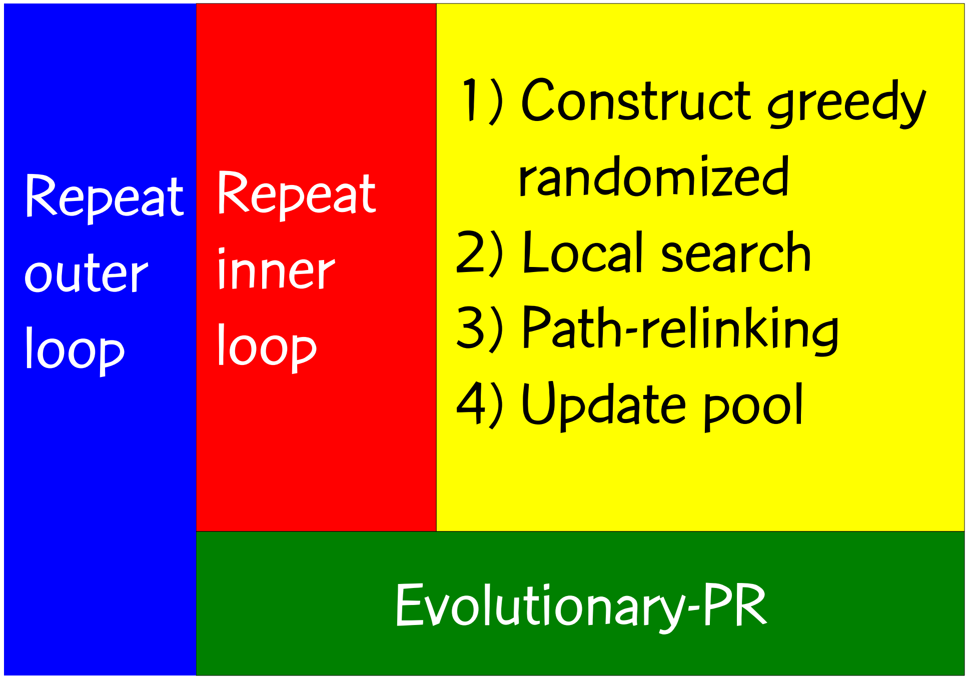


Repeat GRASP with PR loop

1) Construct greedy randomized
2) Local search
3) Path-relinking
4) Update pool

Evolutionary-PR

During GRASP + PR



Repeat outer loop

Repeat inner loop

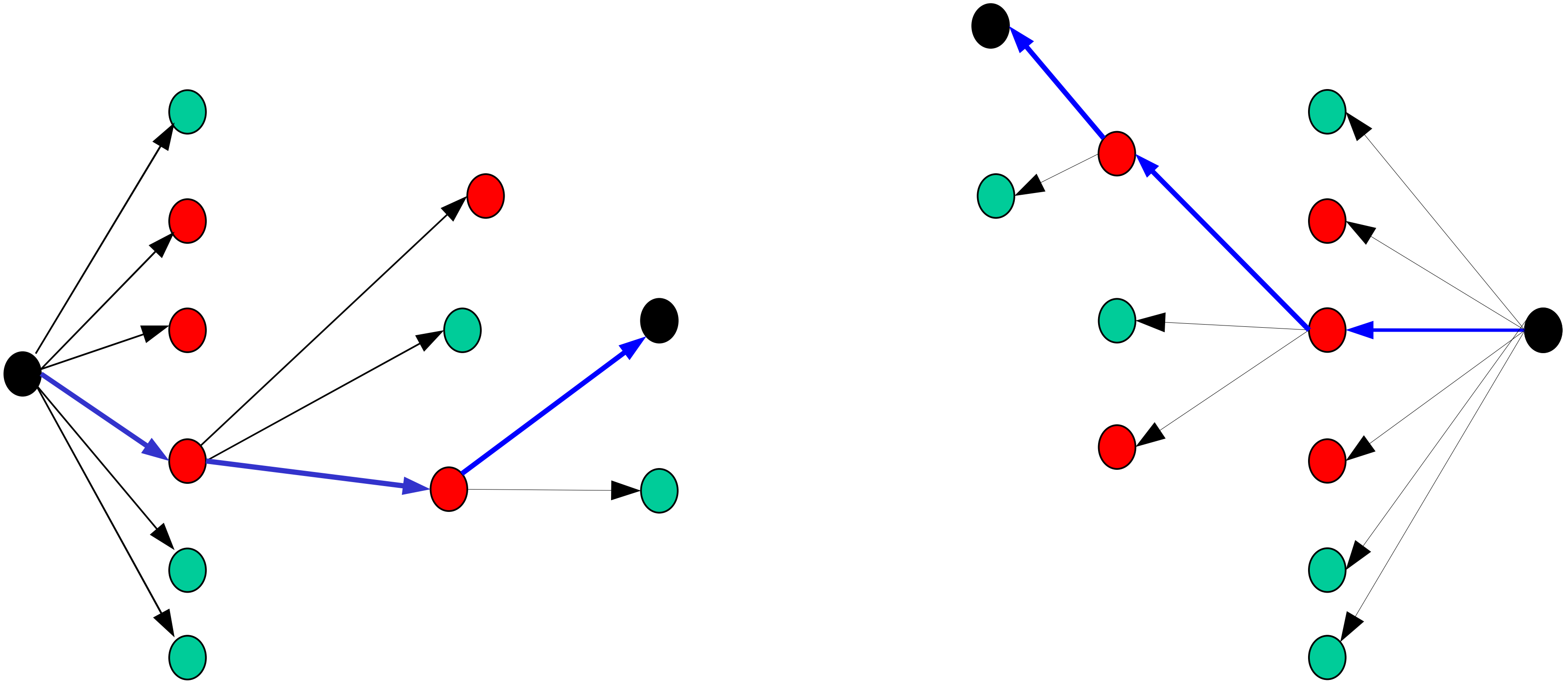
1) Construct greedy randomized
2) Local search
3) Path-relinking
4) Update pool

Evolutionary-PR

(Resende & Werneck, 2004, 2006)

GRASP with EvPR: Implementation ideas

Truncated mixed graPR

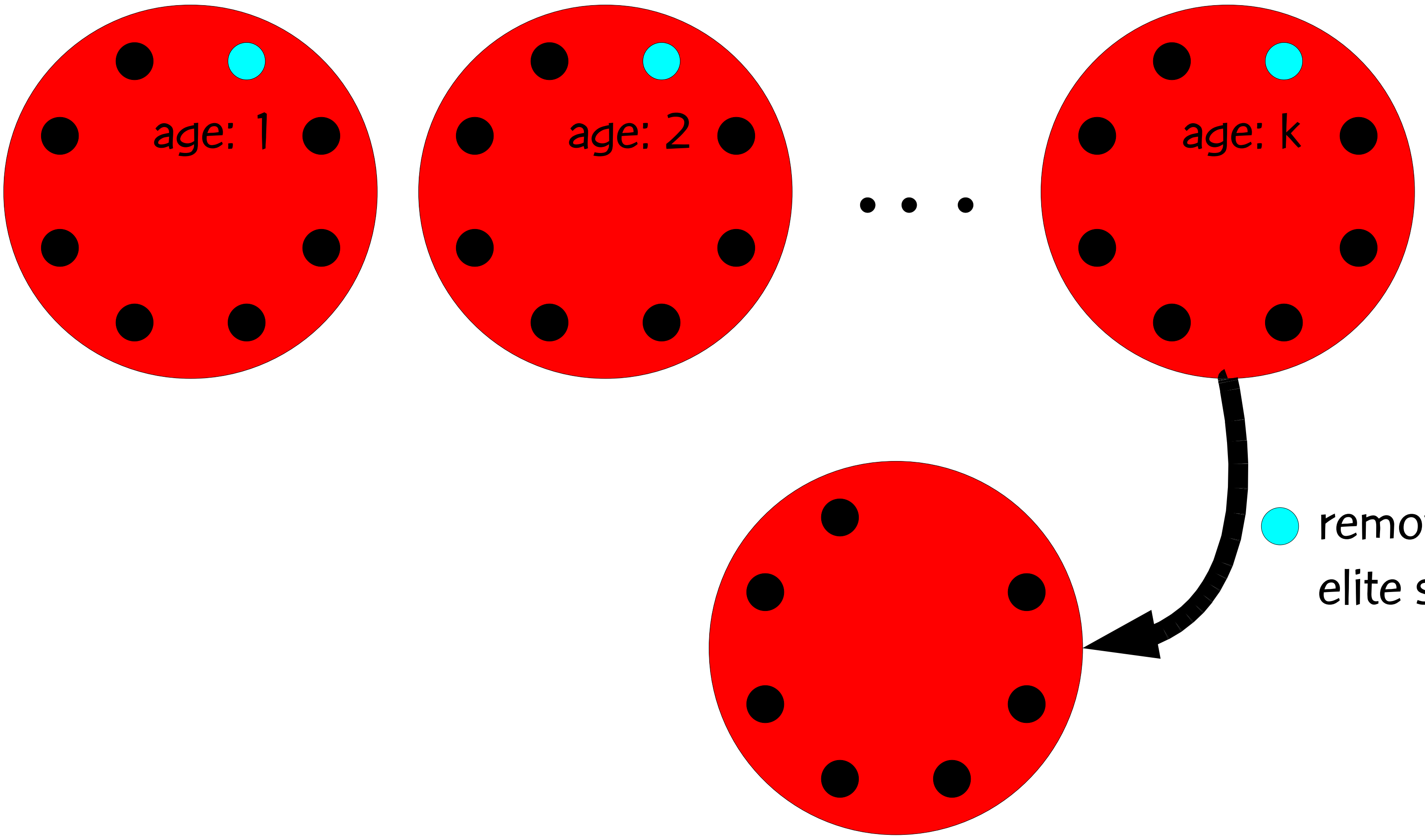


In PR and EvPR, apply one iteration of graPR.

For (x,y), different calls to graPR(x,y) explore different paths.

GRASP with EvPR: Implementation ideas

Force old low-quality elite solutions out

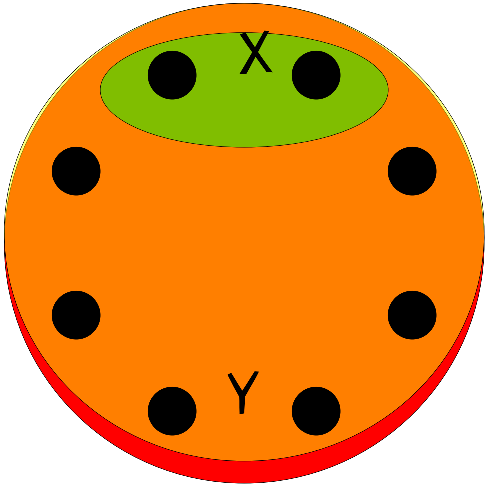


● removed from elite set.

GRASP with EvPR: Implementation ideas

Make set X small and with best pool solutions.

Make set Y be entire pool.



Use set X of size 1 or 2.

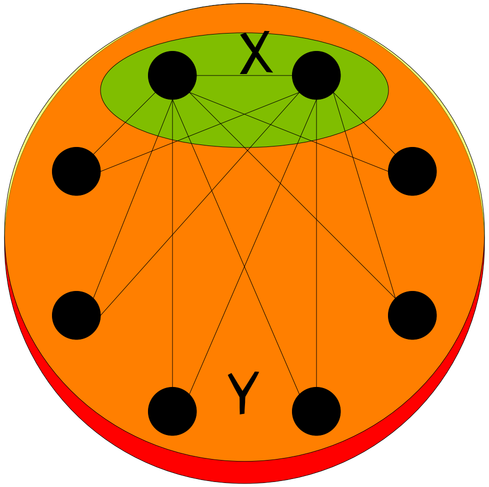
Speeds up EvPR.

Avoids unfruitful calls to graPR(x,y)

GRASP with EvPR: Implementation ideas

Make set X small and with best pool solutions.

Make set Y be entire pool.

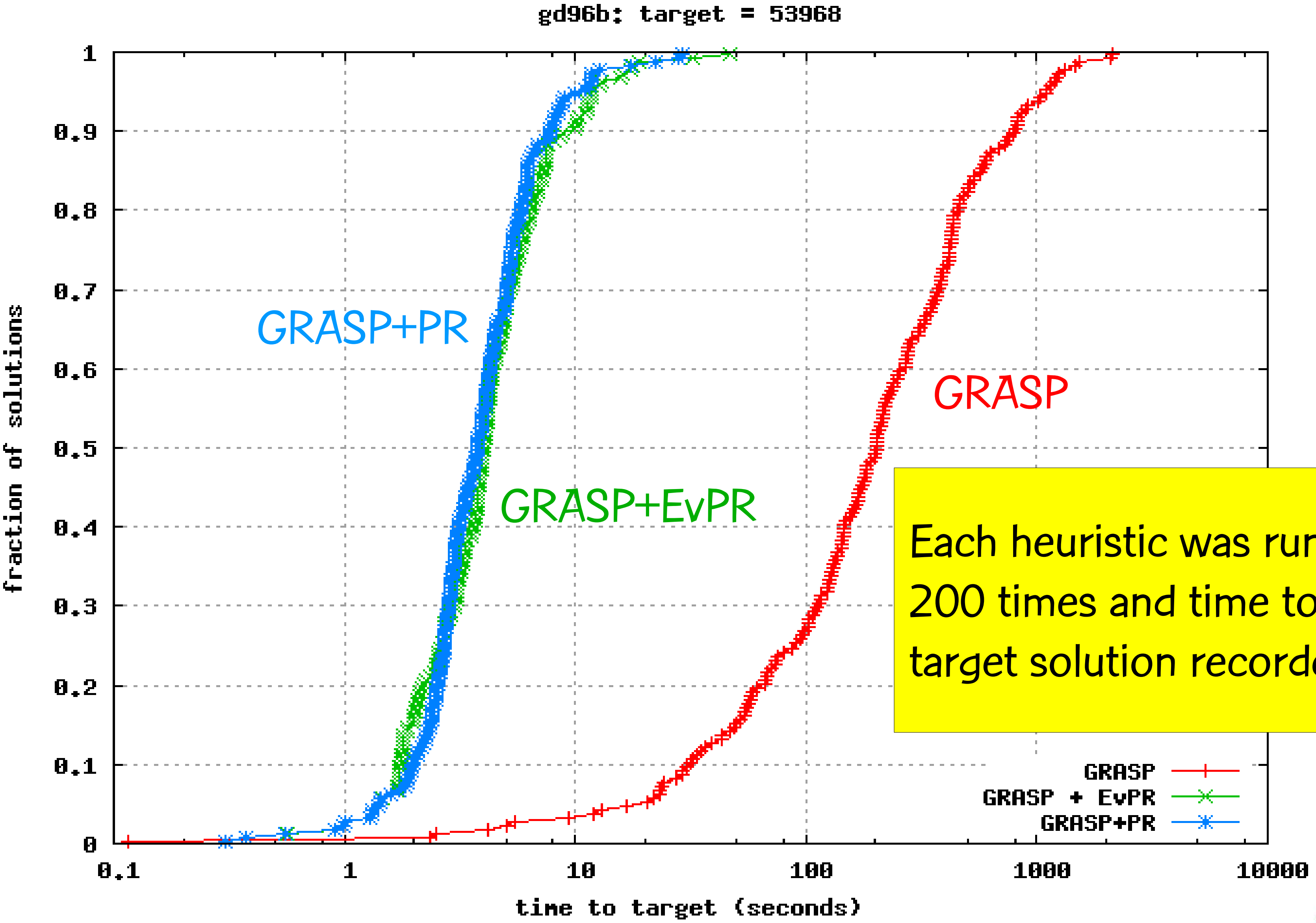


Use set X of size 1 or 2.

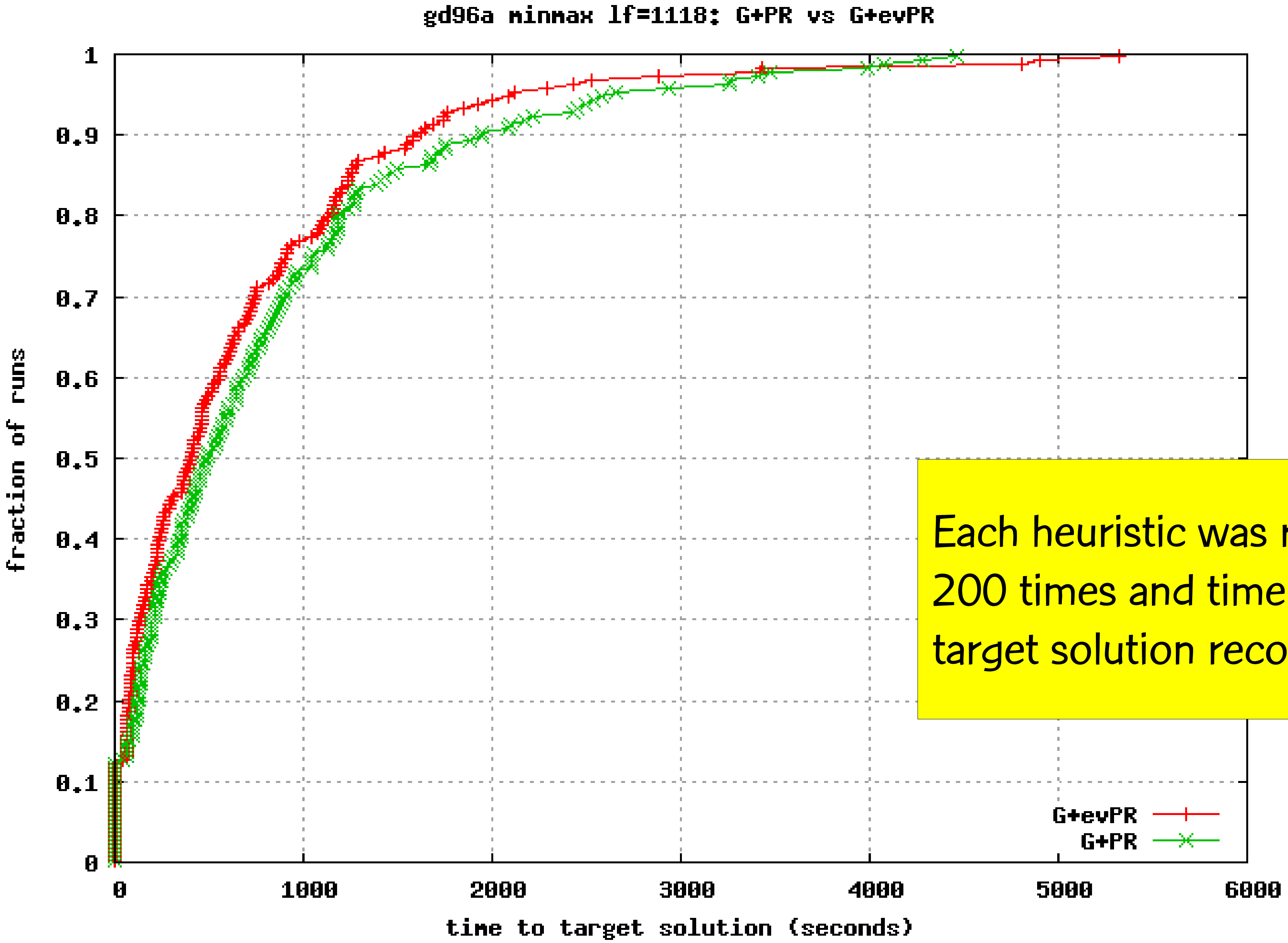
Speeds up EvPR.

Avoids unfruitful calls to graPR(x,y)

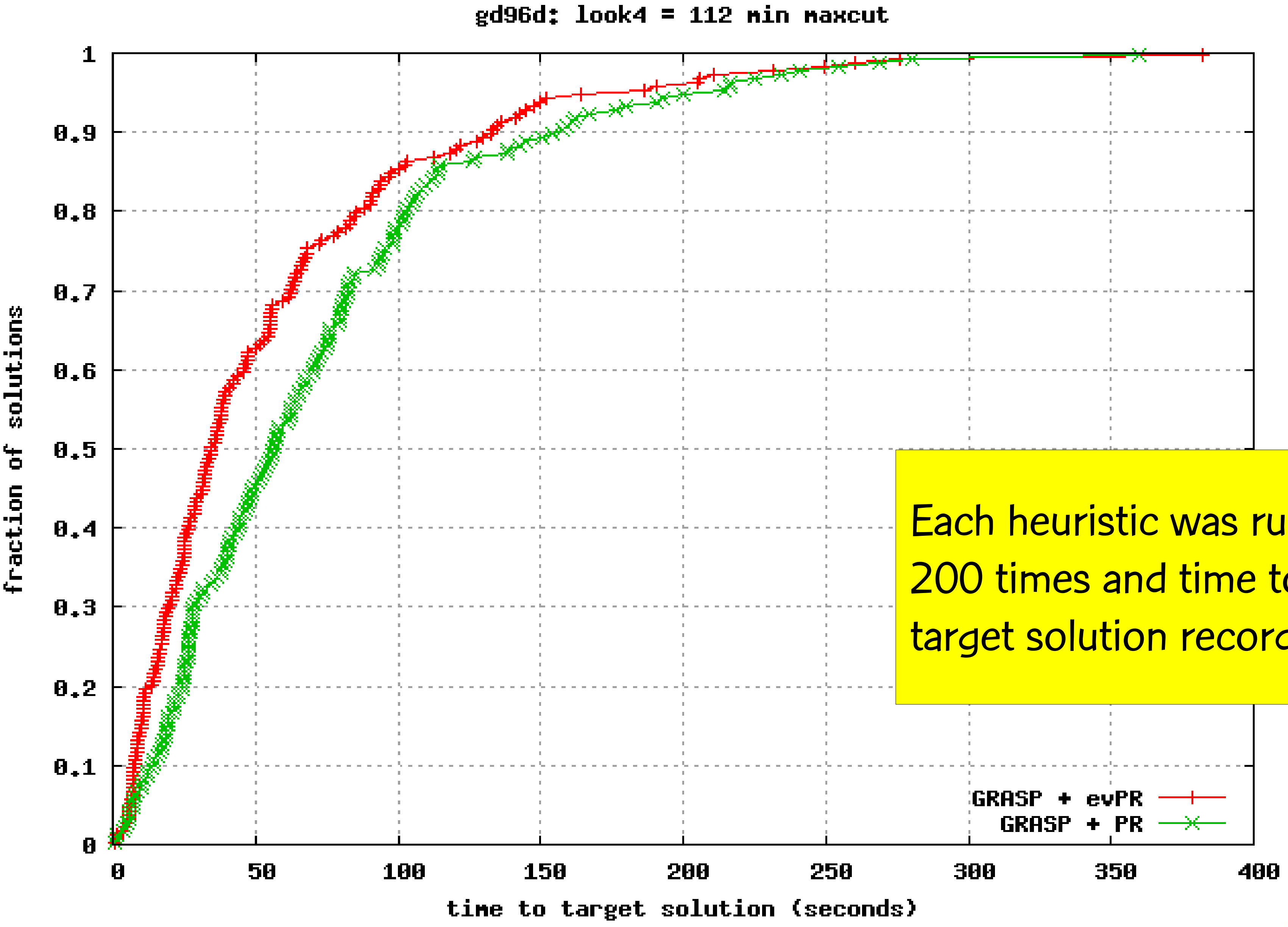
Weights uniformly distributed in interval [1,100]: min sum cuts



Each heuristic was run 200 times and time to target solution recorded.

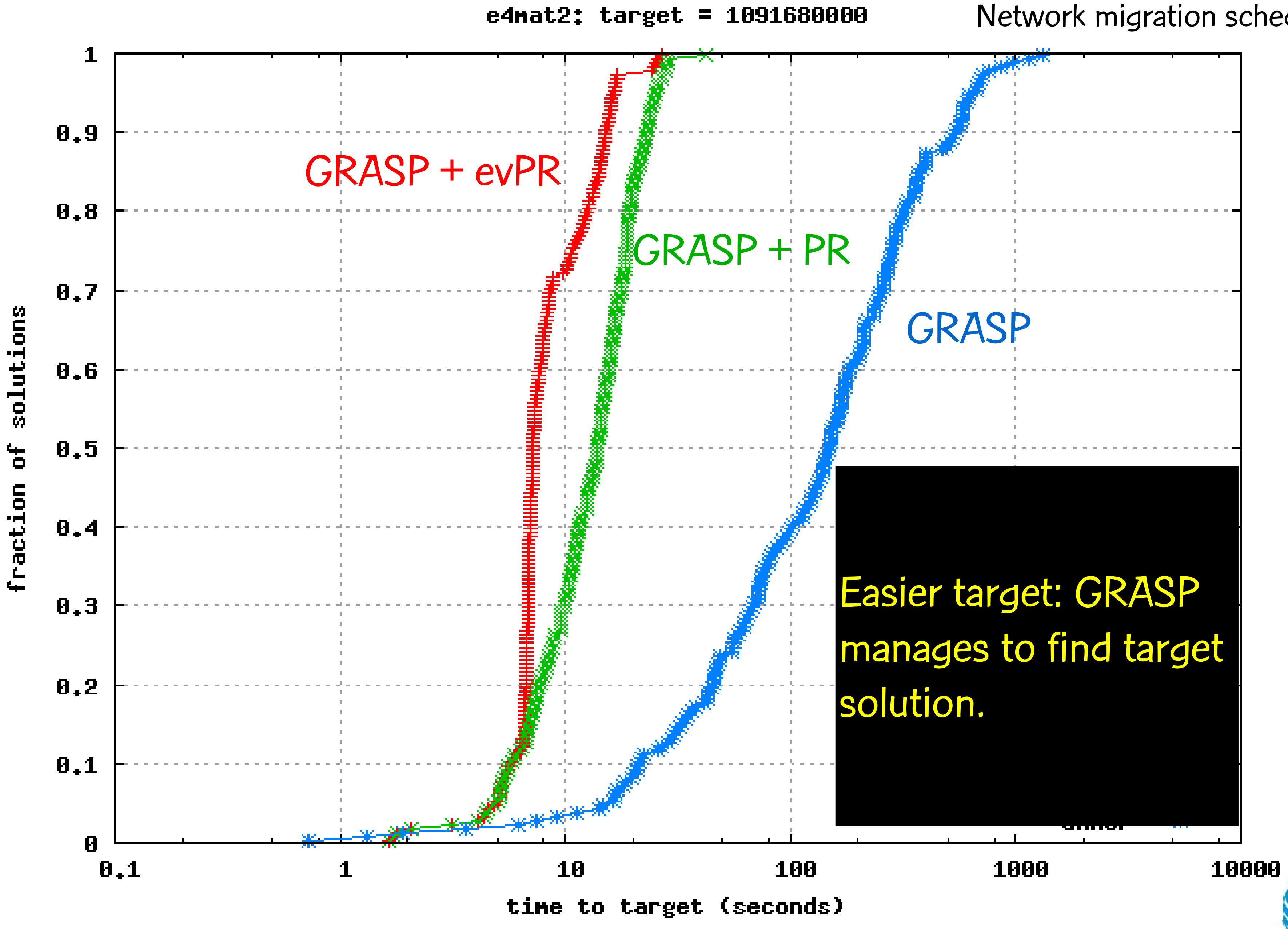


Each heuristic was run 200 times and time to target solution recorded.



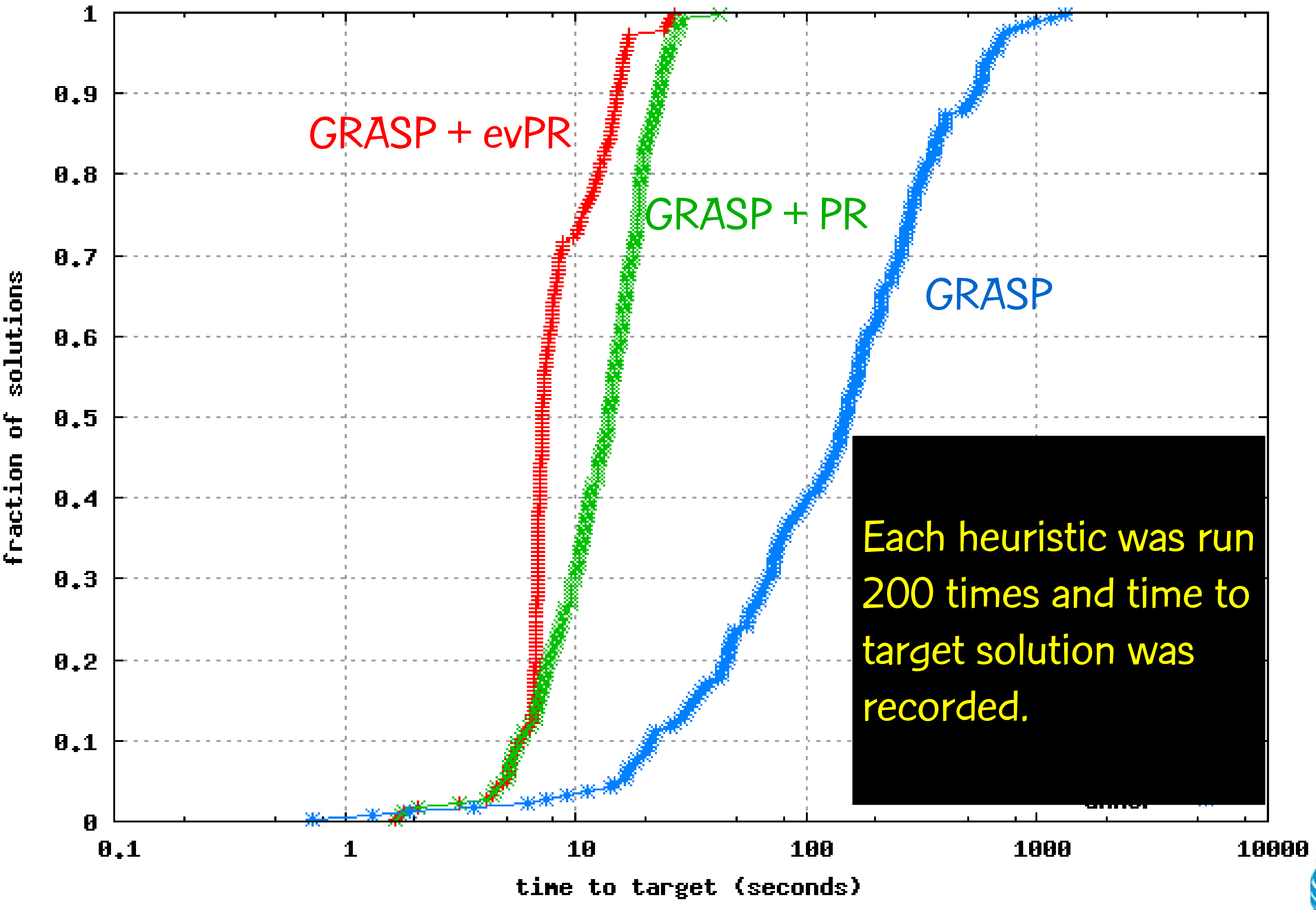
Each heuristic was run 200 times and time to target solution recorded.

Network migration scheduling



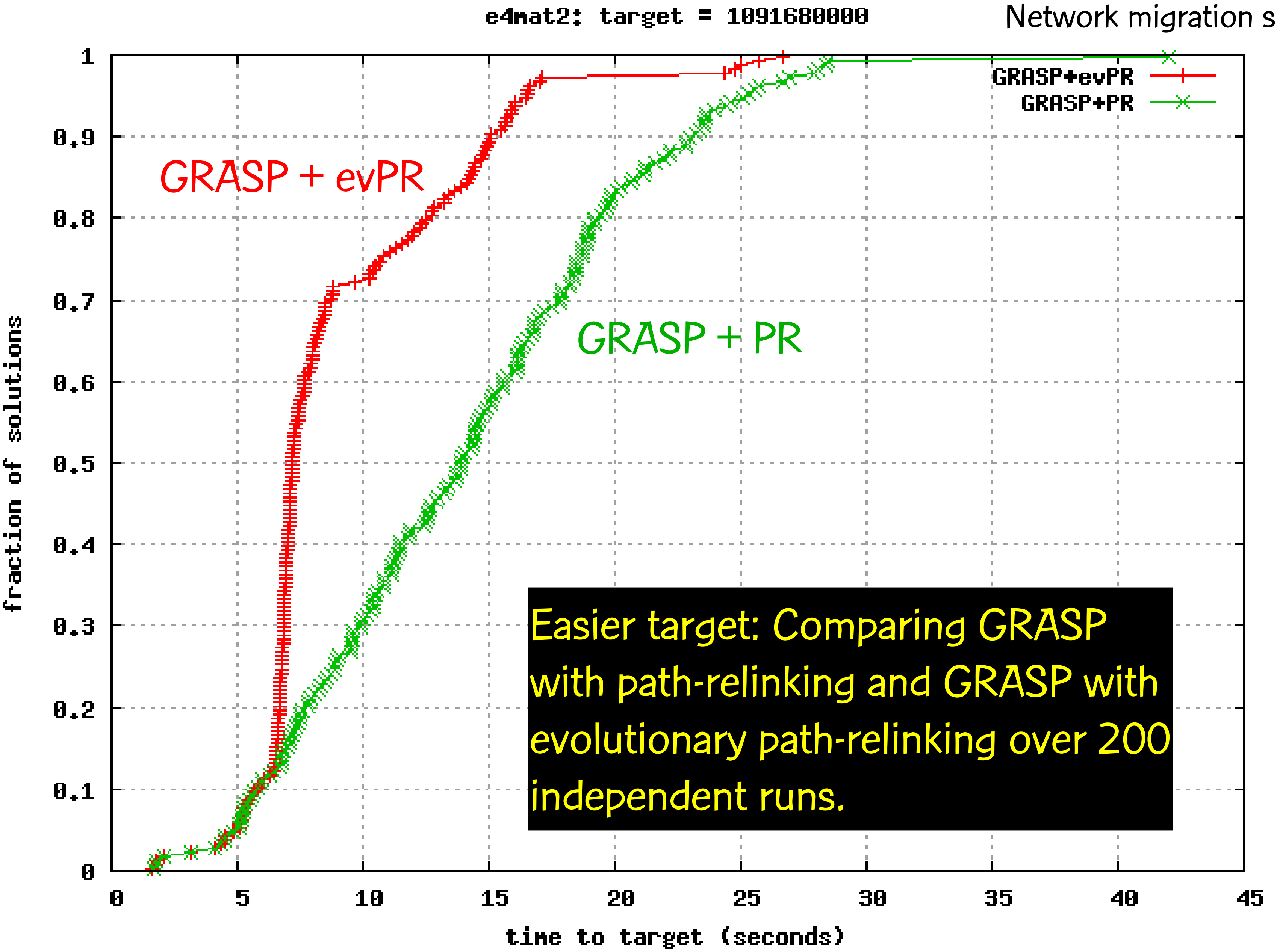
Easier target: GRASP manages to find target solution.

Network migration scheduling

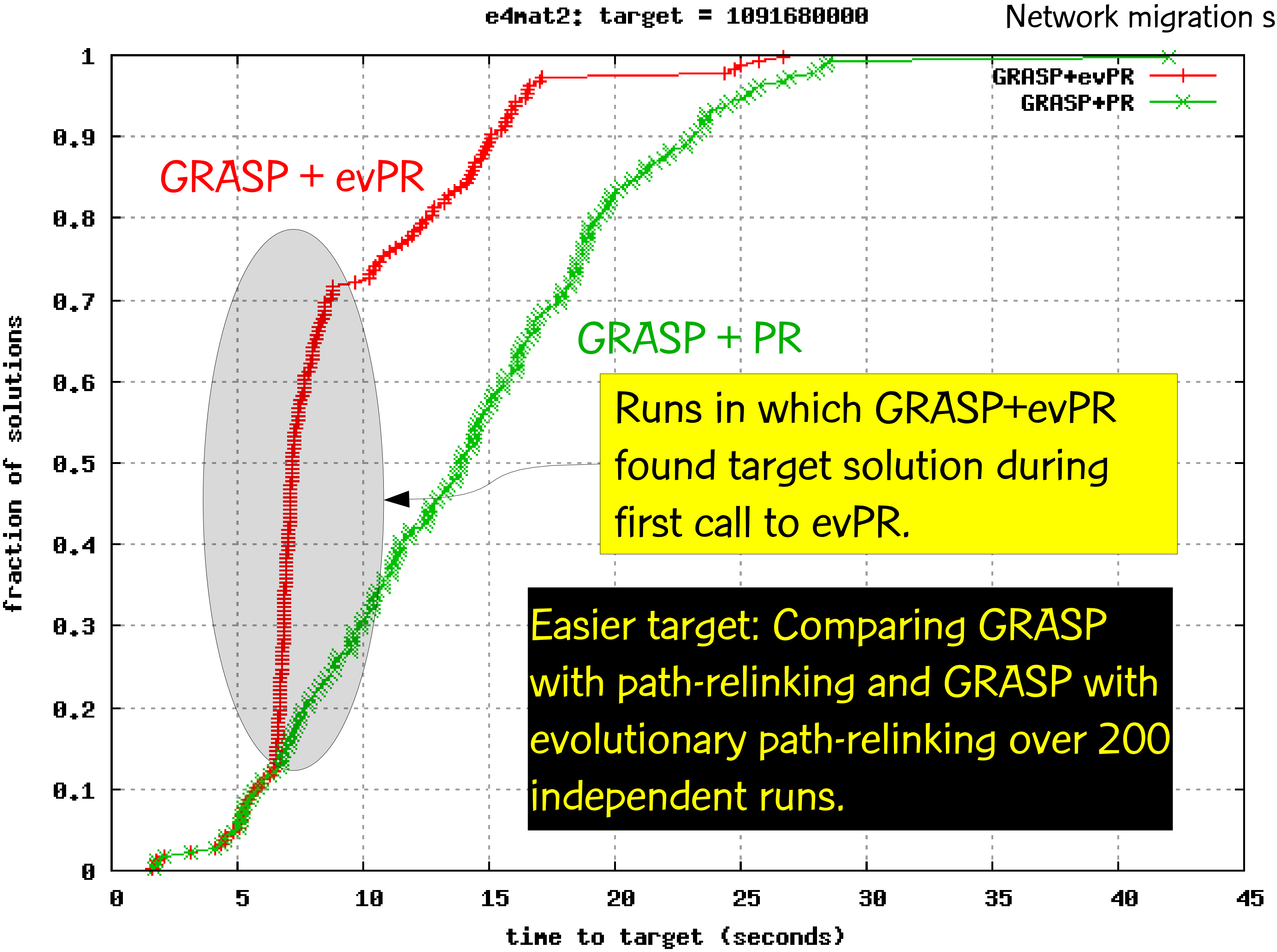


Each heuristic was run 200 times and time to target solution was recorded.

Network migration scheduling



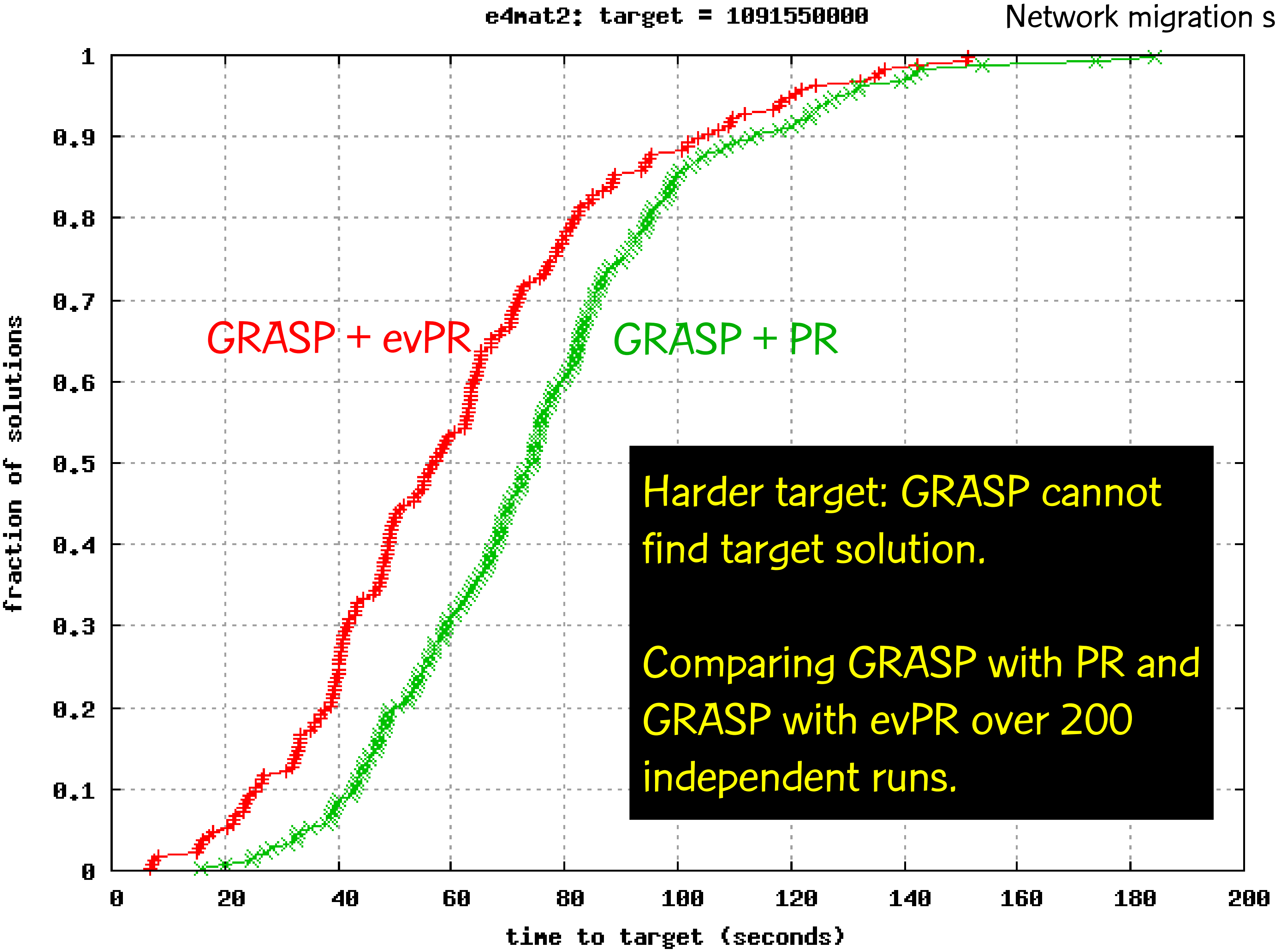
Easier target: Comparing GRASP with path-relinking and GRASP with evolutionary path-relinking over 200 independent runs.



Runs in which GRASP+evPR found target solution during first call to evPR.

Easier target: Comparing GRASP with path-relinking and GRASP with evolutionary path-relinking over 200 independent runs.

Network migration scheduling



Harder target: GRASP cannot find target solution.

Comparing GRASP with PR and GRASP with evPR over 200 independent runs.

Parallel GRASP
and
GRASP with Path-relinking

Solution time distribution

- Proposition: Let $P(t,p)$ be the probability of not having found a given target solution value in t time units with p independent processors.

  If $P(t,1) = \exp[-(t-\mu)/\lambda]$, with non-negative λ and μ (two-parameter exponential distribution), then

  $P(t,p) = \exp[-p.(t-\mu)/\lambda]$.

  $\Rightarrow$ if $p\mu \ll \lambda$, then the probability of finding a solution within a given target value in time $p \times t$ with a sequential algorithm is approximately equal to that of finding a solution with the same quality in time t with p processors.

Solution time distribution

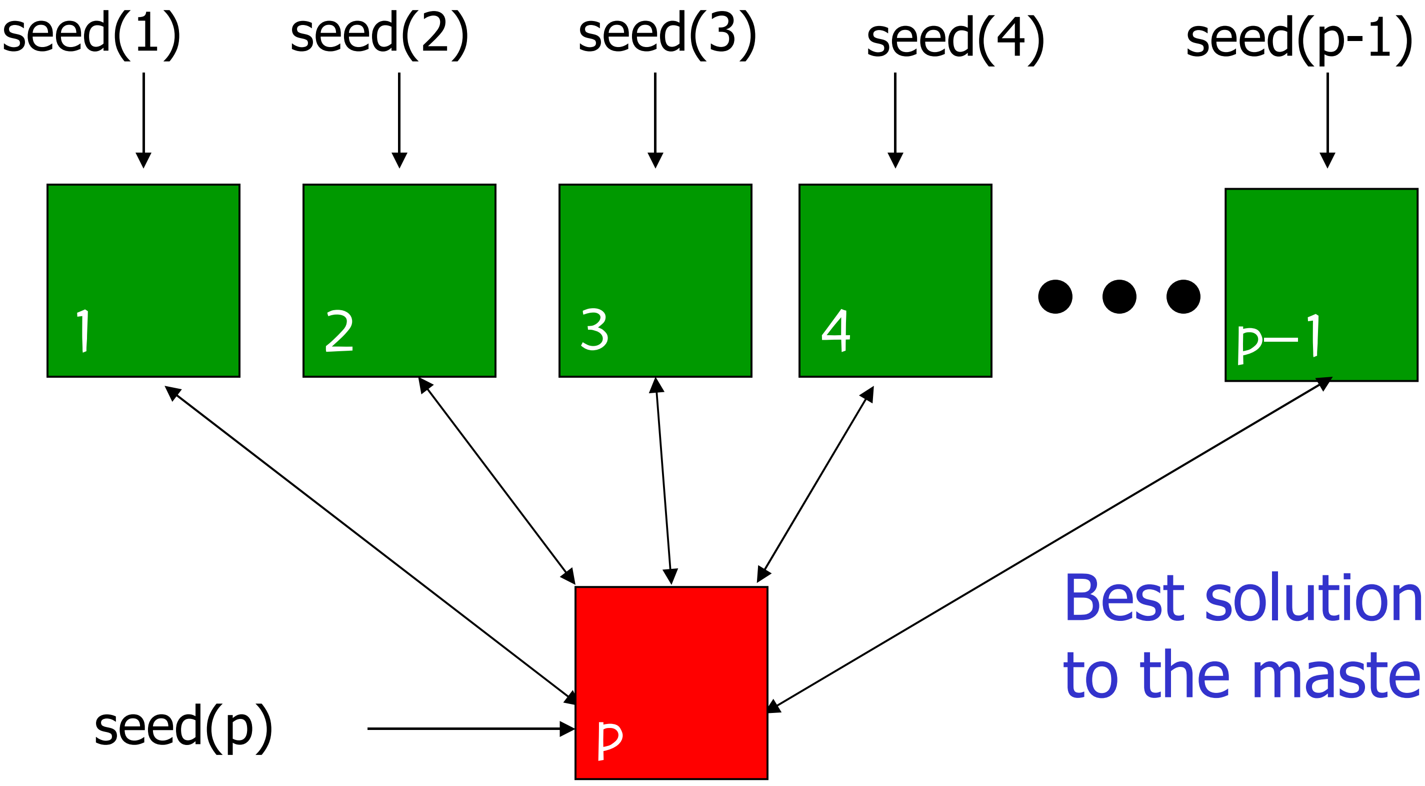
GRASP and GRASP with path-relinking have running times whose distributions fit a shifted exponential distribution.

Therefore, one should expect approximate linear speedup in a straightforward (independent) parallel implementation.

Parallel independent implementation

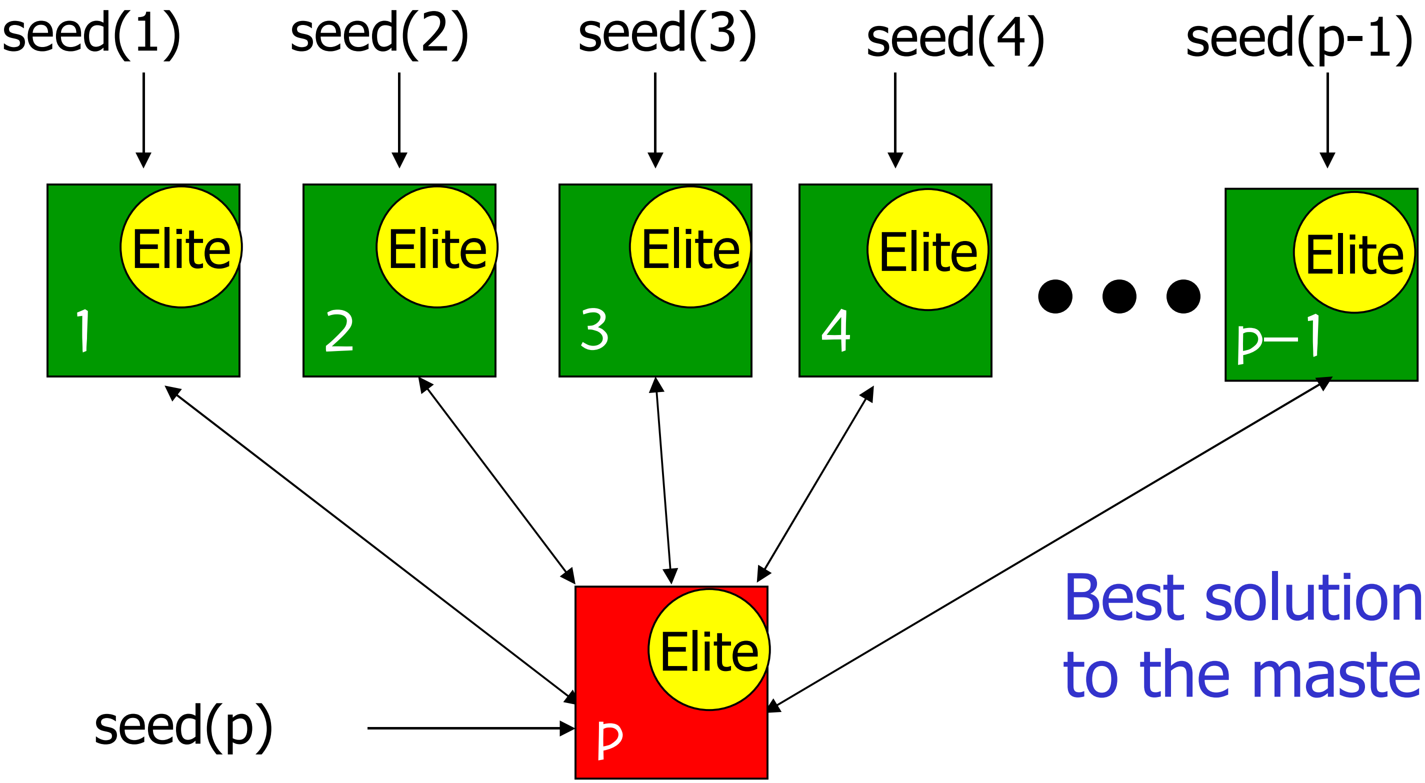
- Parallelism in metaheuristics: robustness
  Duni-Eksioglu, Pardalos, and Resende (2002)
- Multiple-walk independent-thread strategy:
  - p processors available
  - Iterations evenly distributed over p processors
  - Each processor keeps a copy of data and algorithms.
  - One processor acts as the master handling seeds, data, and iteration counter, besides performing GRASP iterations.
  - Each processor performs Max_Iterations/p iterations.

Parallel GRASP independent implementation



Best solution is sent to the master.

Parallel GRASP with PR independent implementation



Best solution is sent to the master.

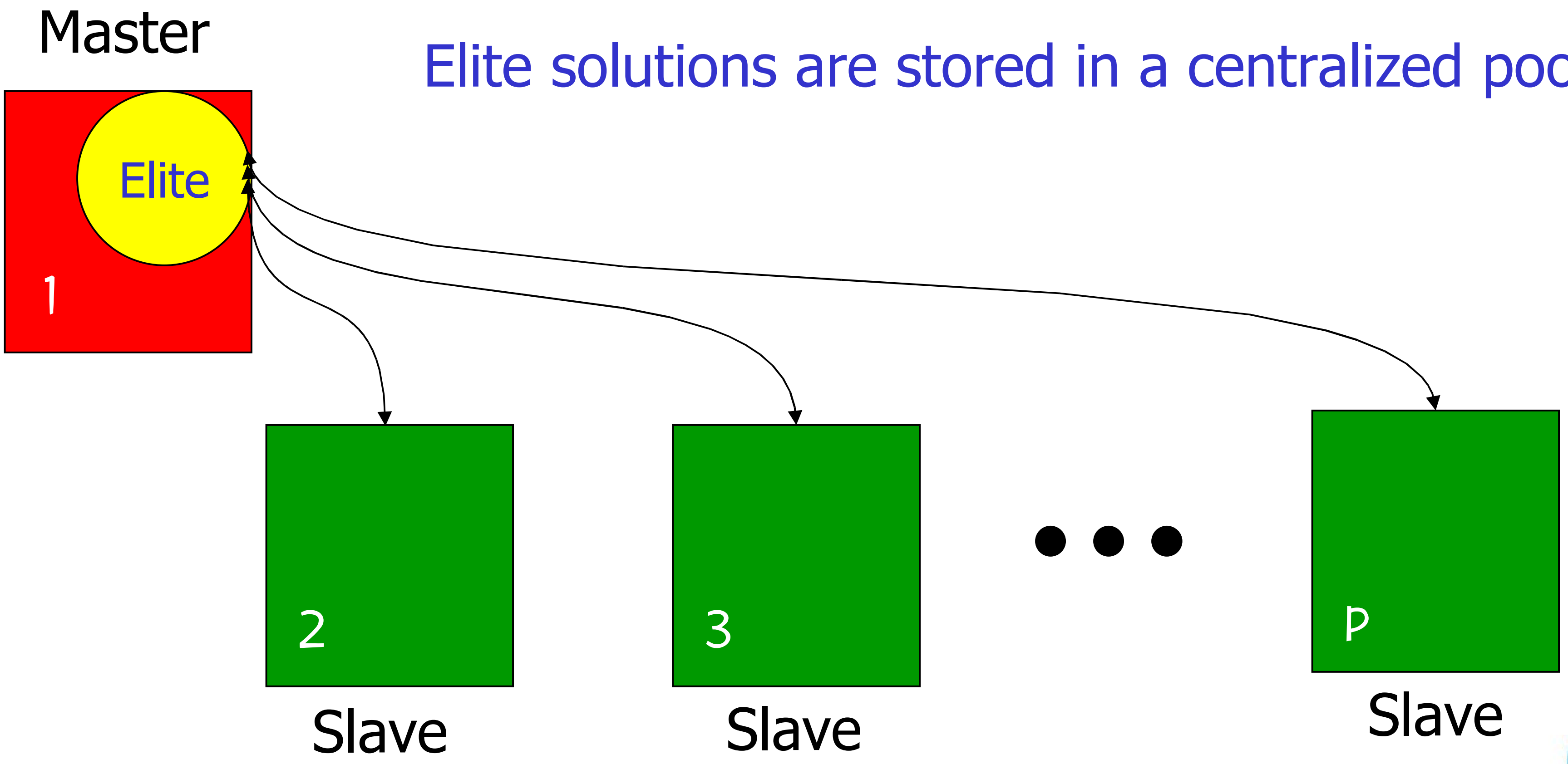
Parallel cooperative GRASP with PR implementations

- Two strategies have been proposed for multiple-walk cooperative-thread parallel implementations:
  - Centralized strategies
  - Distributed strategies

Centralized strategy

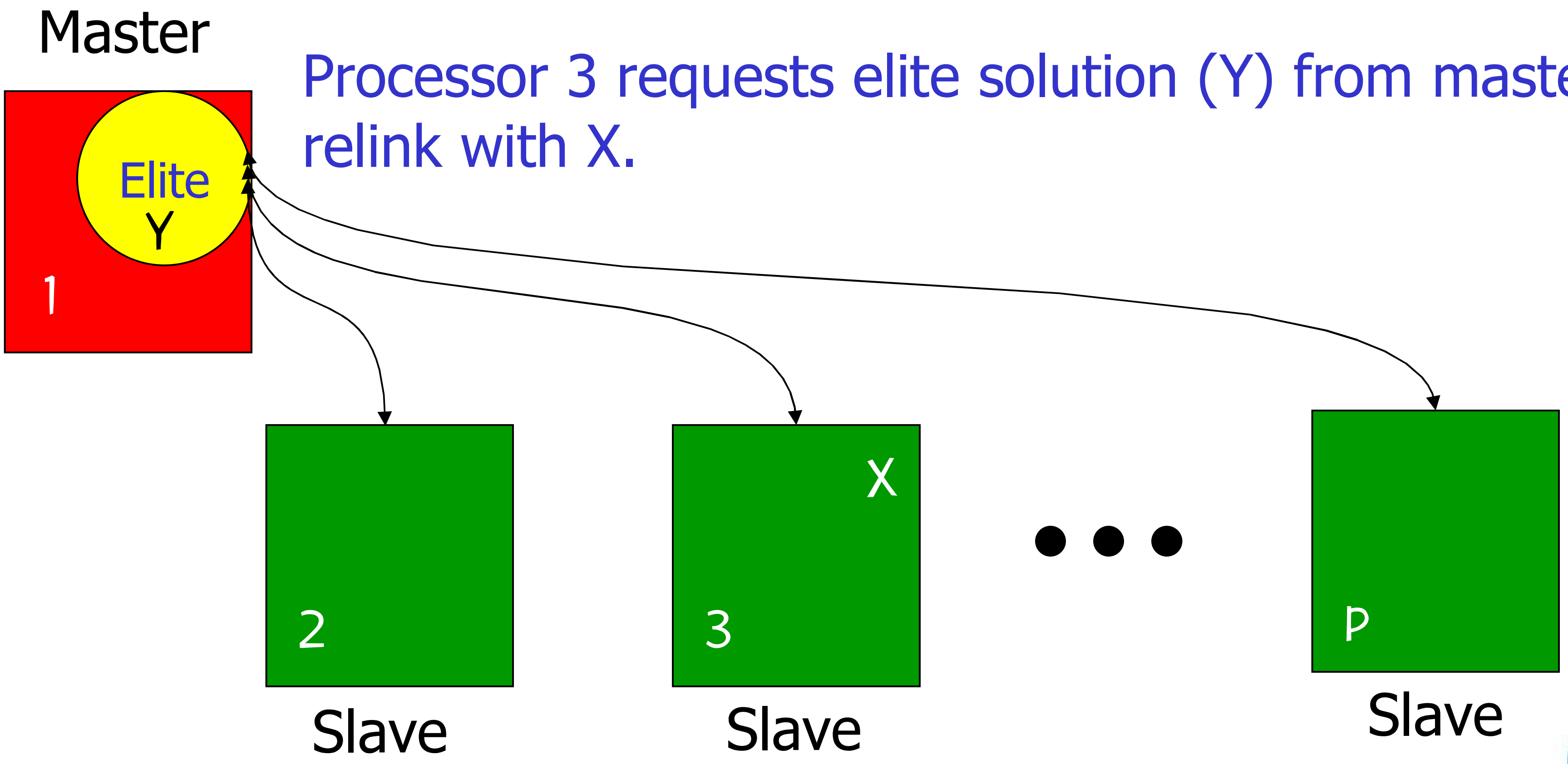
- GRASP construction and local search is performed independently in each processor.
- Elite solution is requested by each processor from centralized pool so processor can do PR.
- Result of PR is sent to pool for testing for insertion.
- Collaboration takes place when elite solution is sent to other processor.

Parallel centralized cooperative strategy



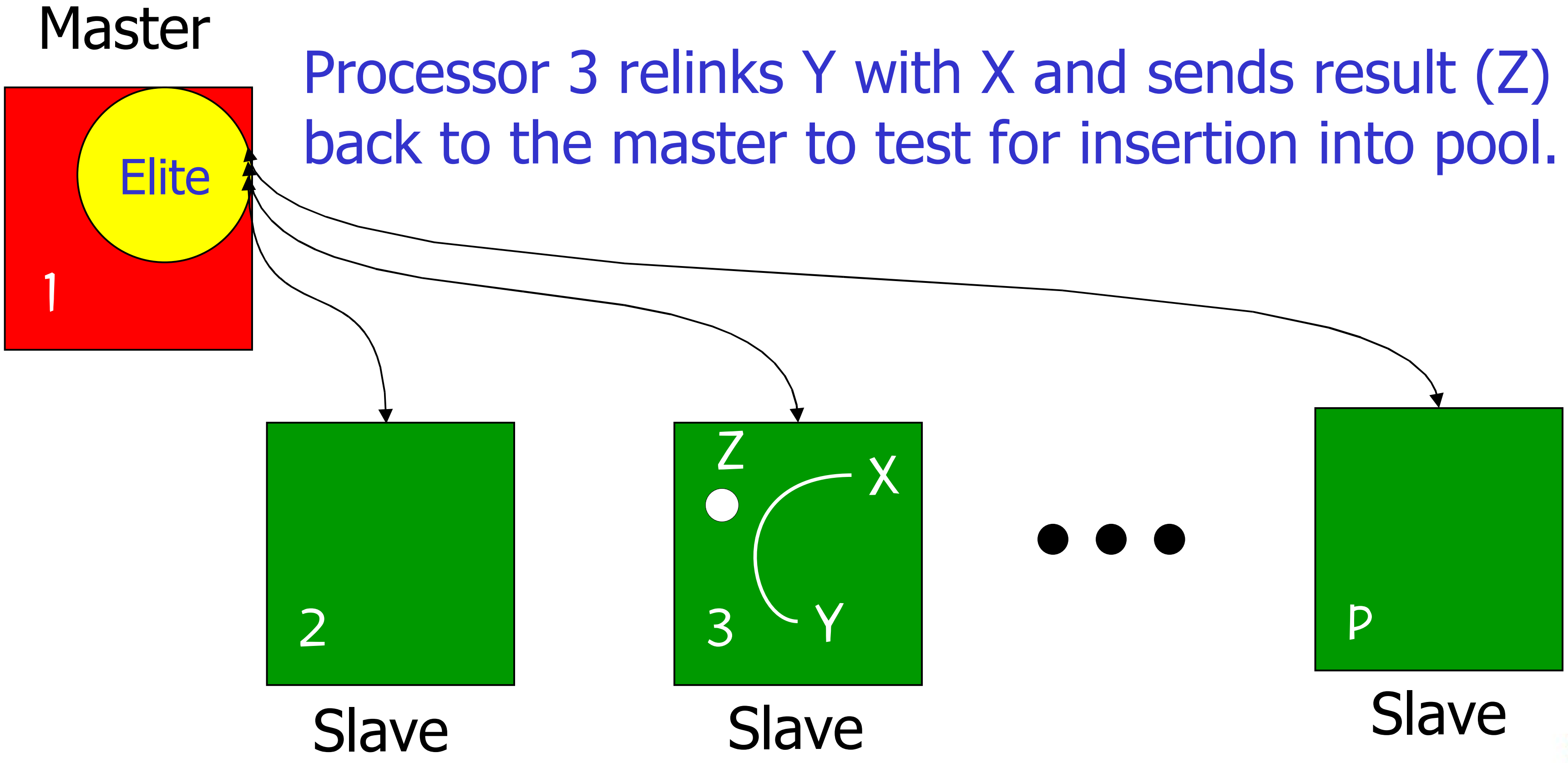
Elite solutions are stored in a centralized pool.

Parallel centralized cooperative strategy



Processor 3 requests elite solution (Y) from master to relink with X.

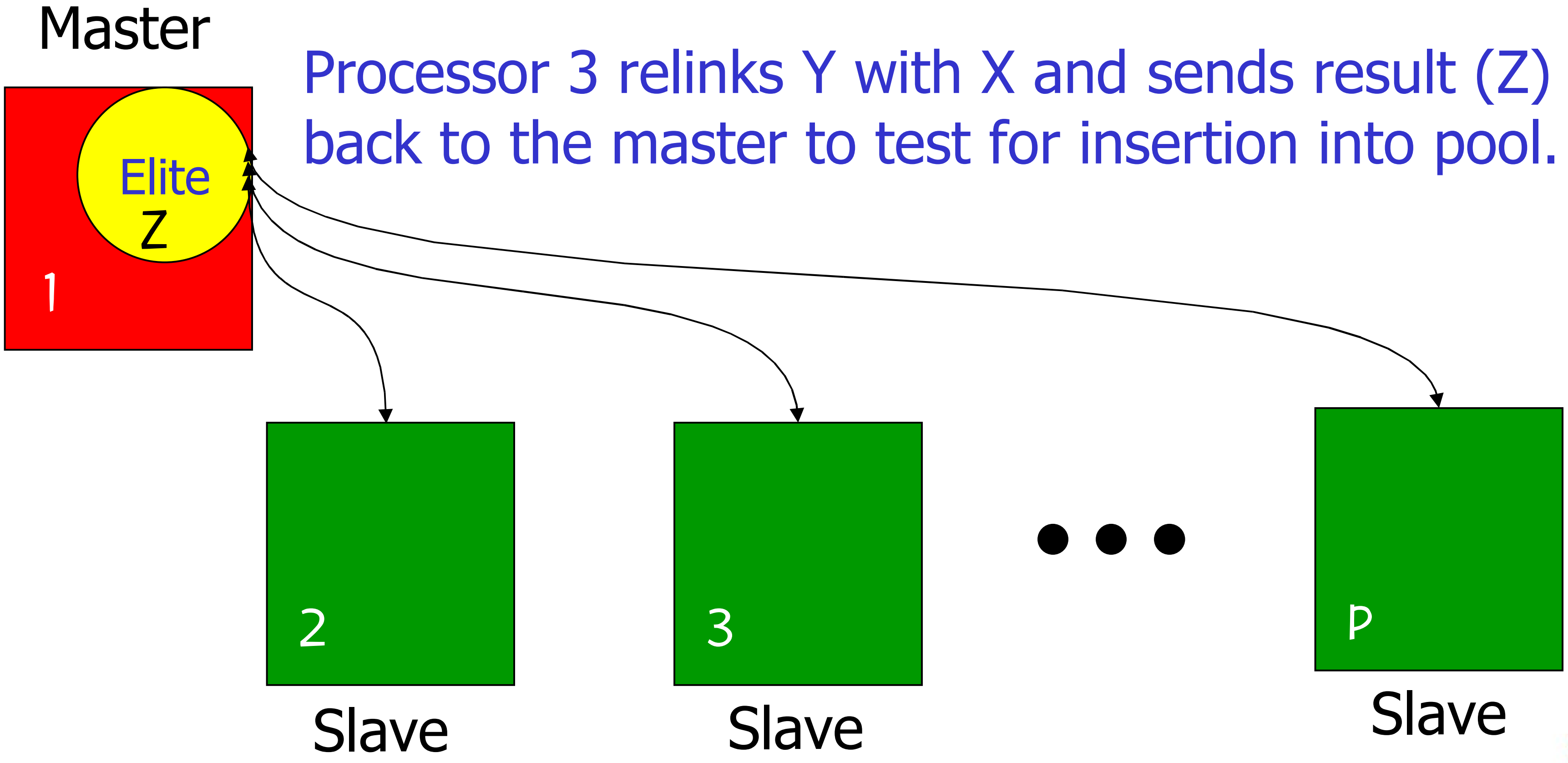
Parallel centralized cooperative strategy



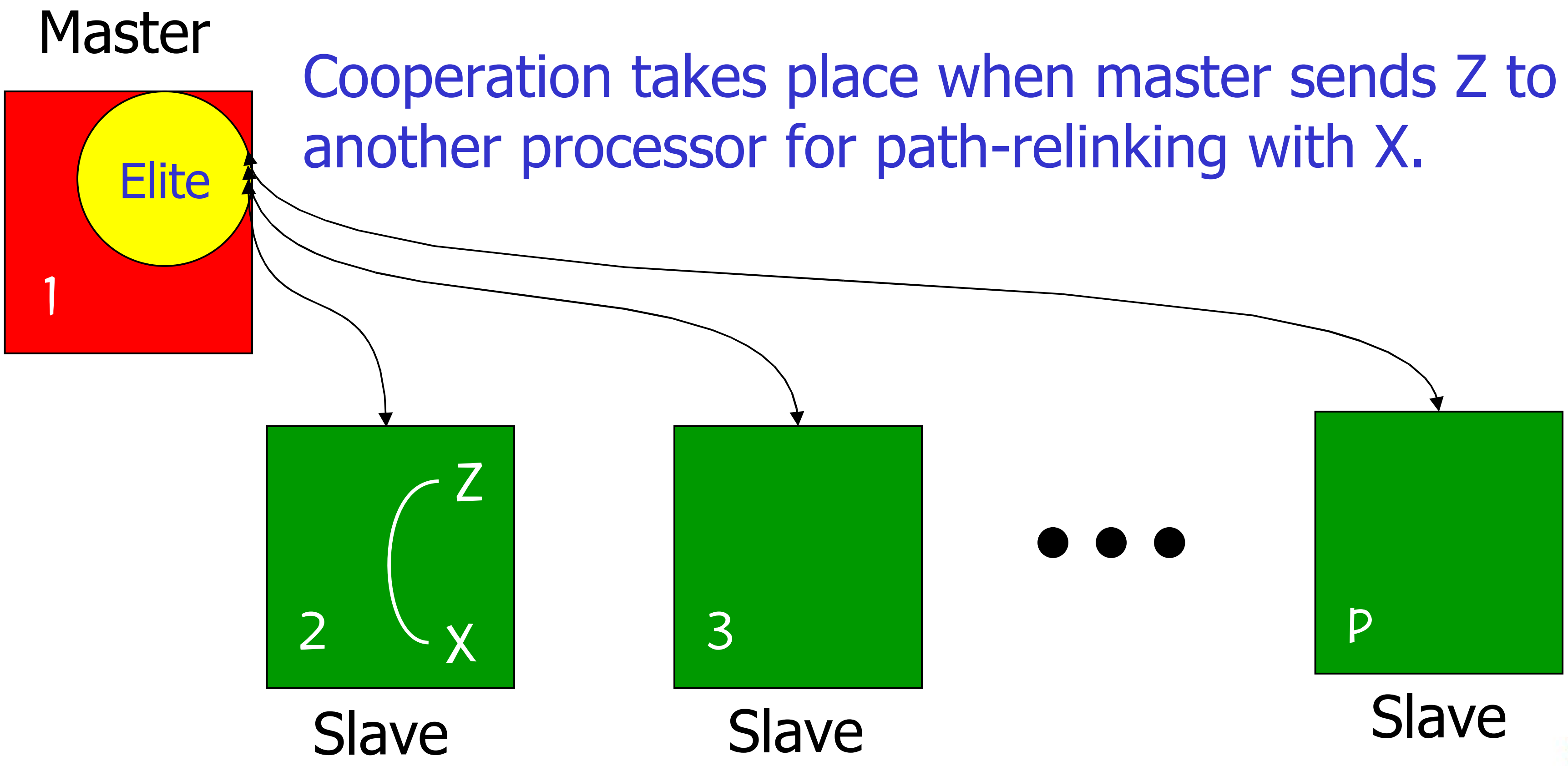
Processor 3 relinks Y with X and sends result (Z) back to the master to test for insertion into pool.

Parallel centralized cooperative strategy

Processor 3 relinks Y with X and sends result (Z) back to the master to test for insertion into pool.



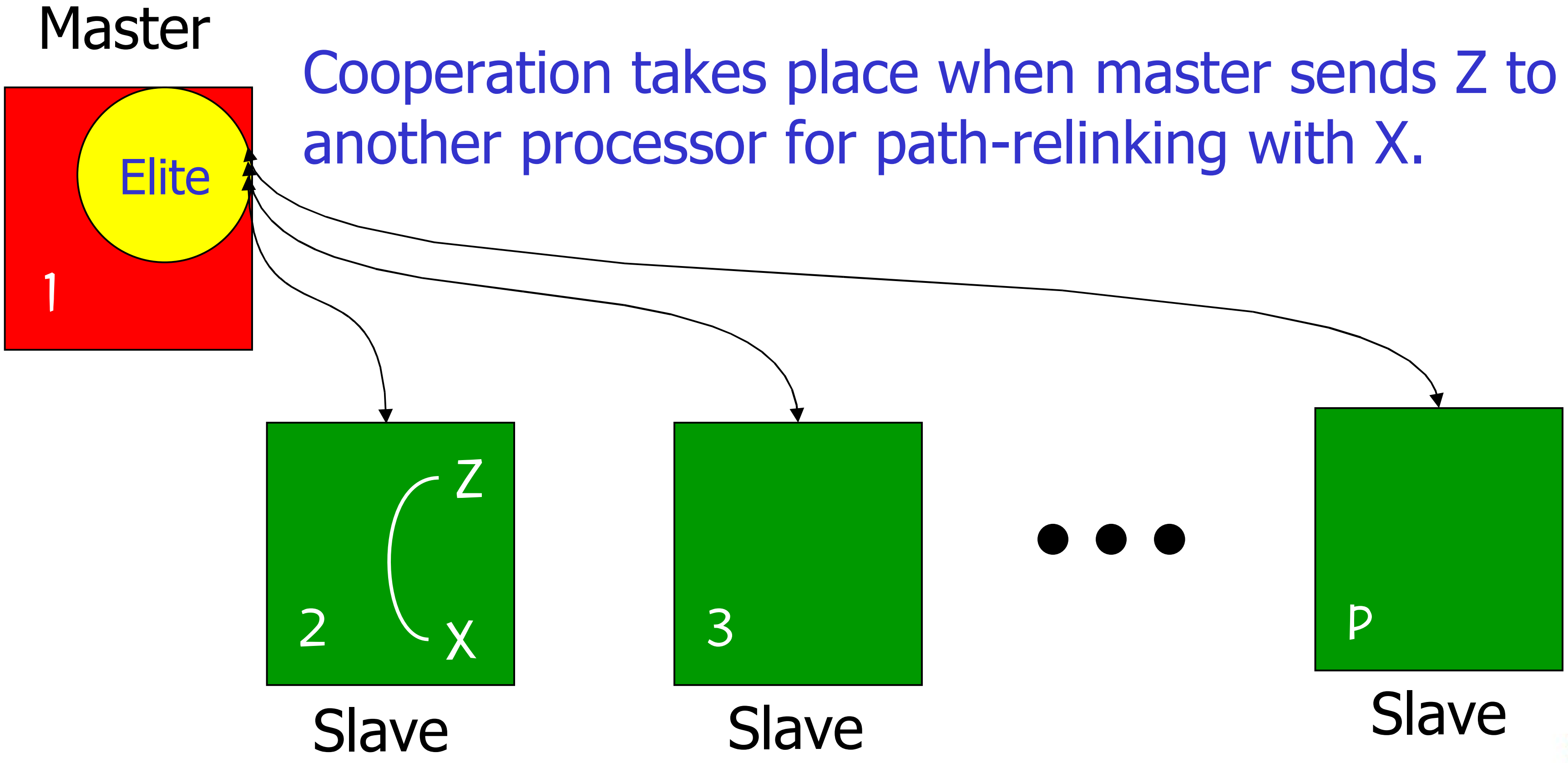
Parallel centralized cooperative strategy



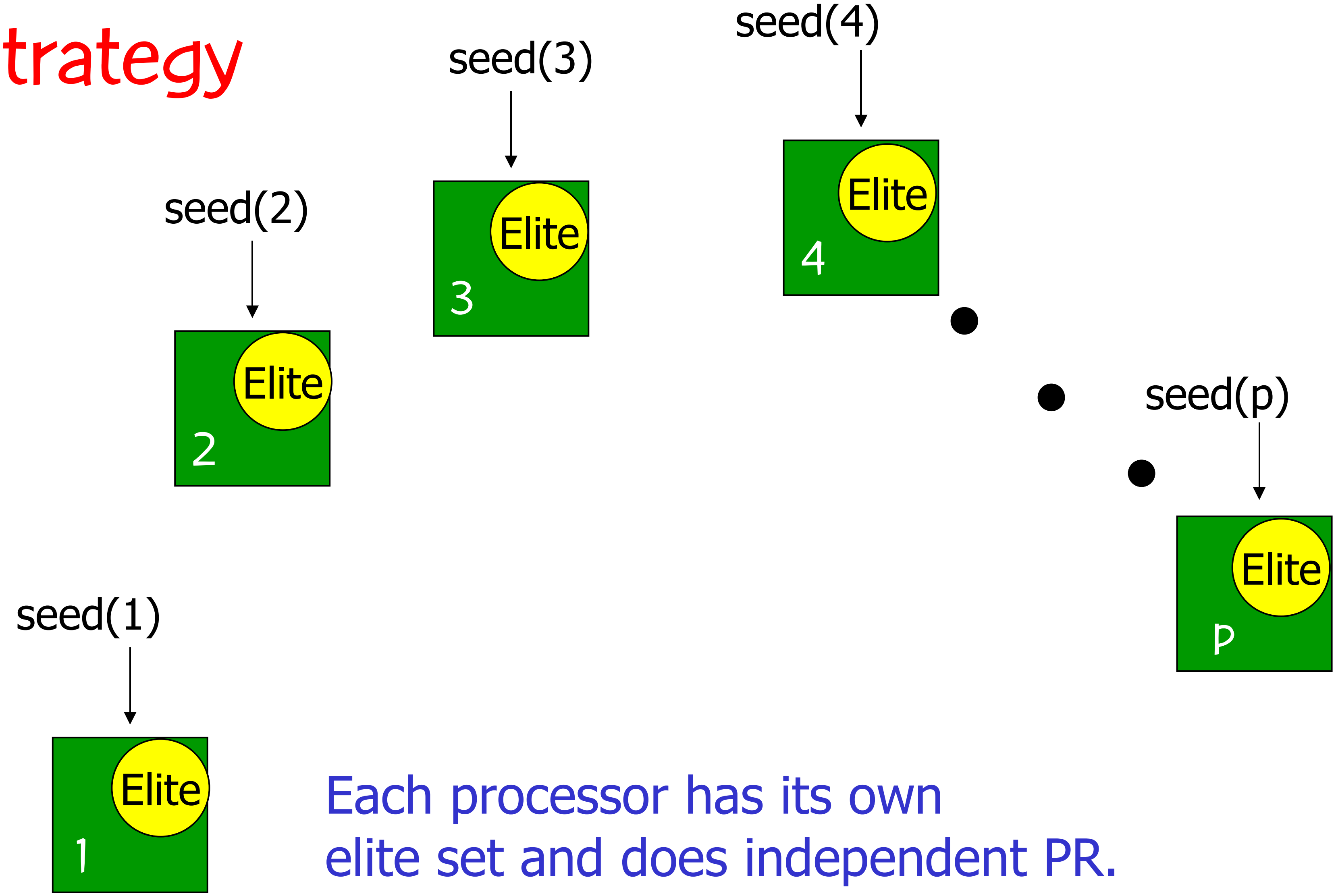
Cooperation takes place when master sends Z to another processor for path-relinking with X.

Parallel distributed cooperative strategy

Cooperation takes place when master sends Z to another processor for path-relinking with X.

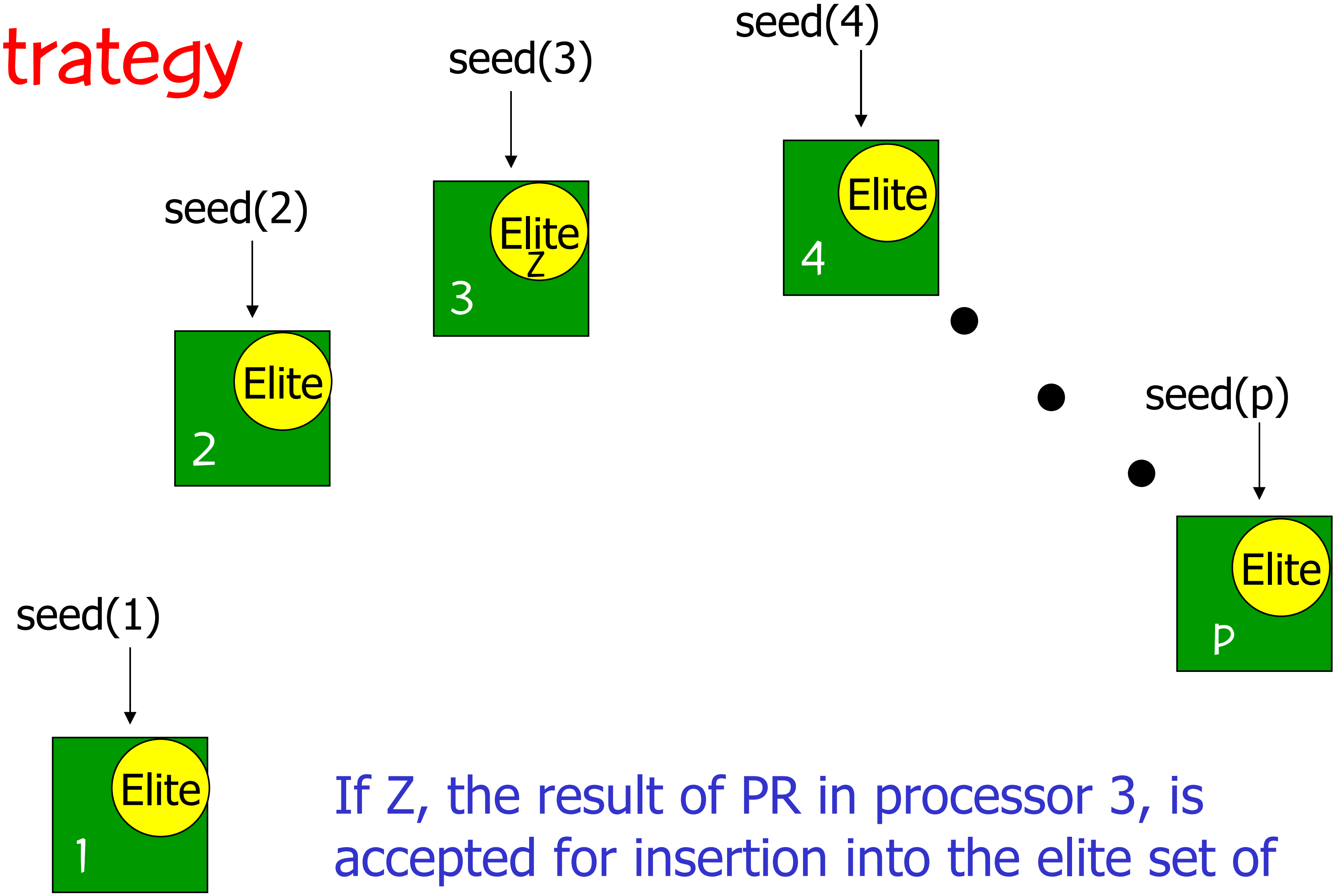


Parallel distributed cooperative strategy



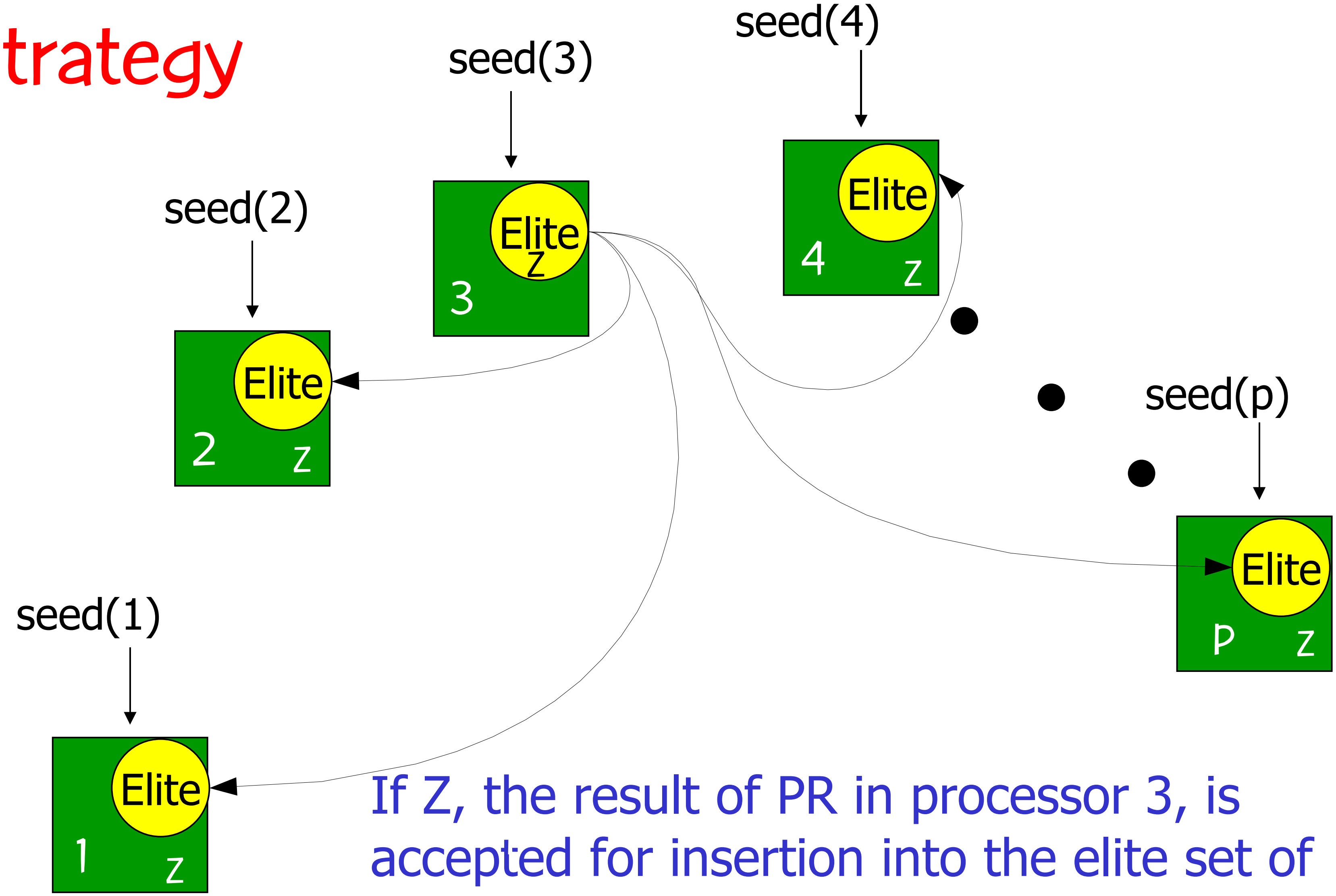
Each processor has its own
elite set and does independent PR.

Parallel distributed cooperative strategy

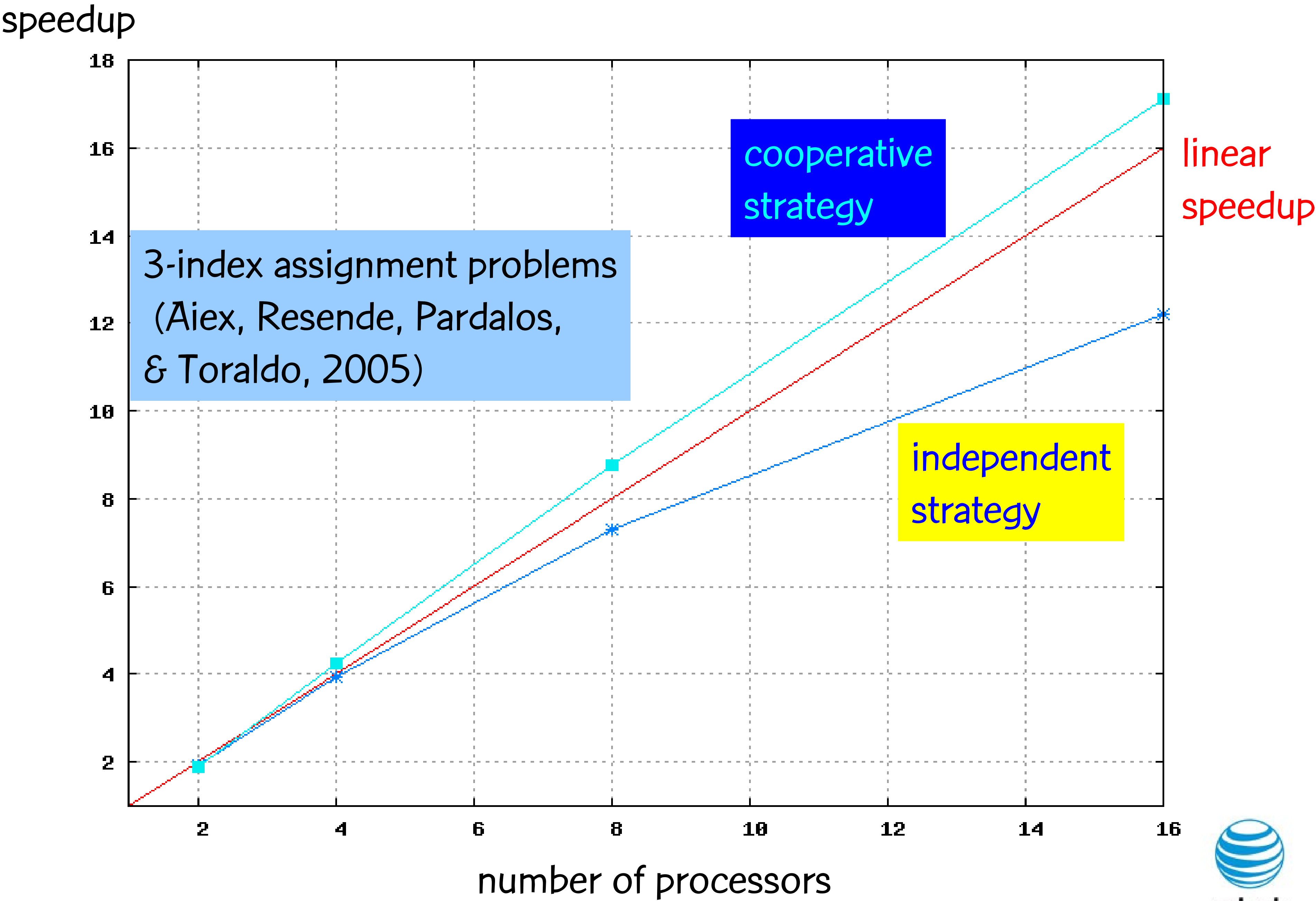


If Z, the result of PR in processor 3, is accepted for insertion into the elite set of processor 3, it is sent for testing in all the other pools.

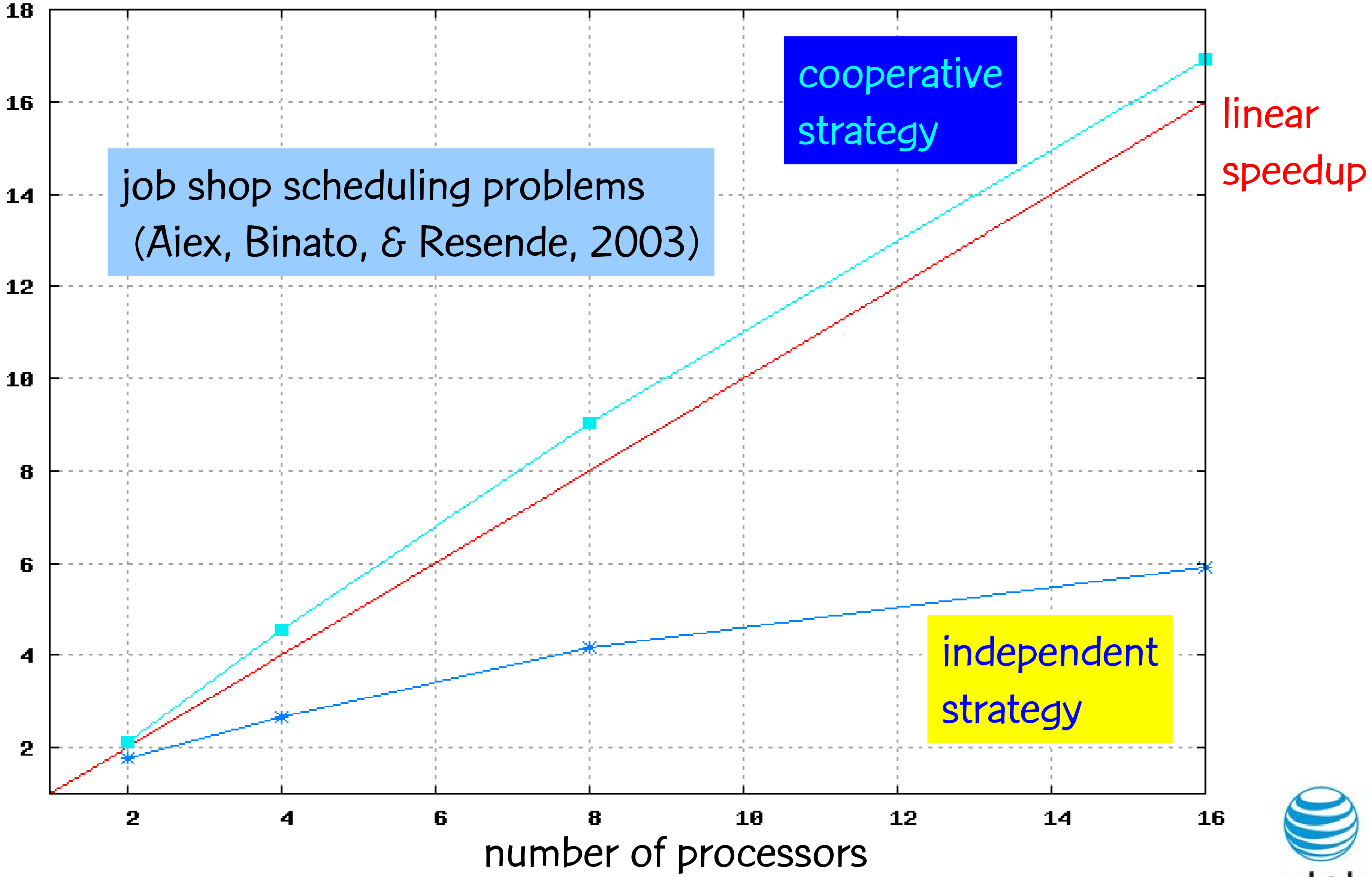
Parallel distributed cooperative strategy



If Z, the result of PR in processor 3, is accepted for insertion into the elite set of processor 3, it is sent for testing in all the other pools.



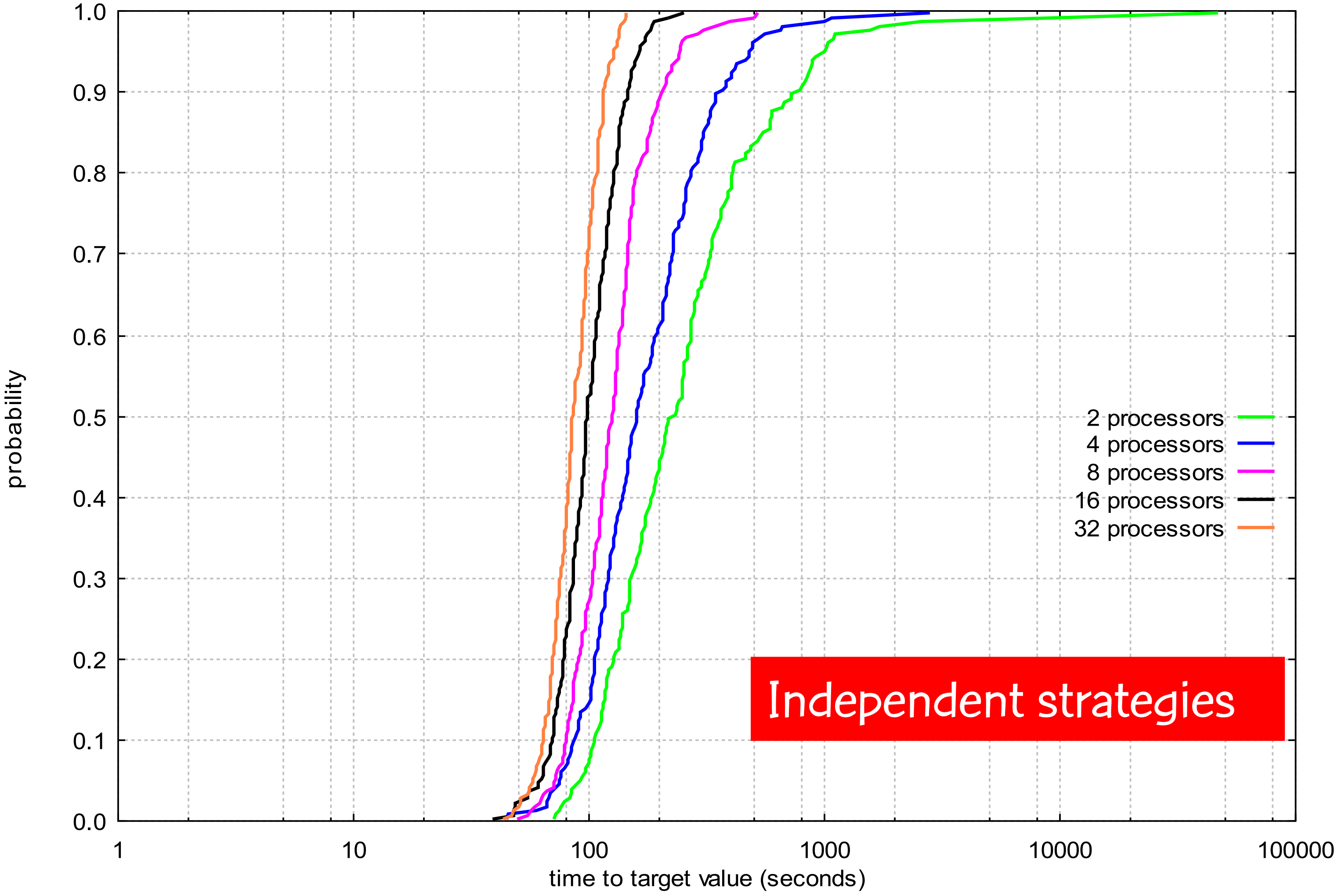
3-index assignment problems (Aiex, Resende, Pardalos, & Toraldo, 2005) speedup


18
16
14
12
10
8
6
4
2
2
4
6
8
10
12
14
16
number of processors
cooperative strategy
linear speedup
independent strategy
job shop scheduling problems
(Aiex, Binato, & Resende, 2003)

Parallel environment at PUC-Rio

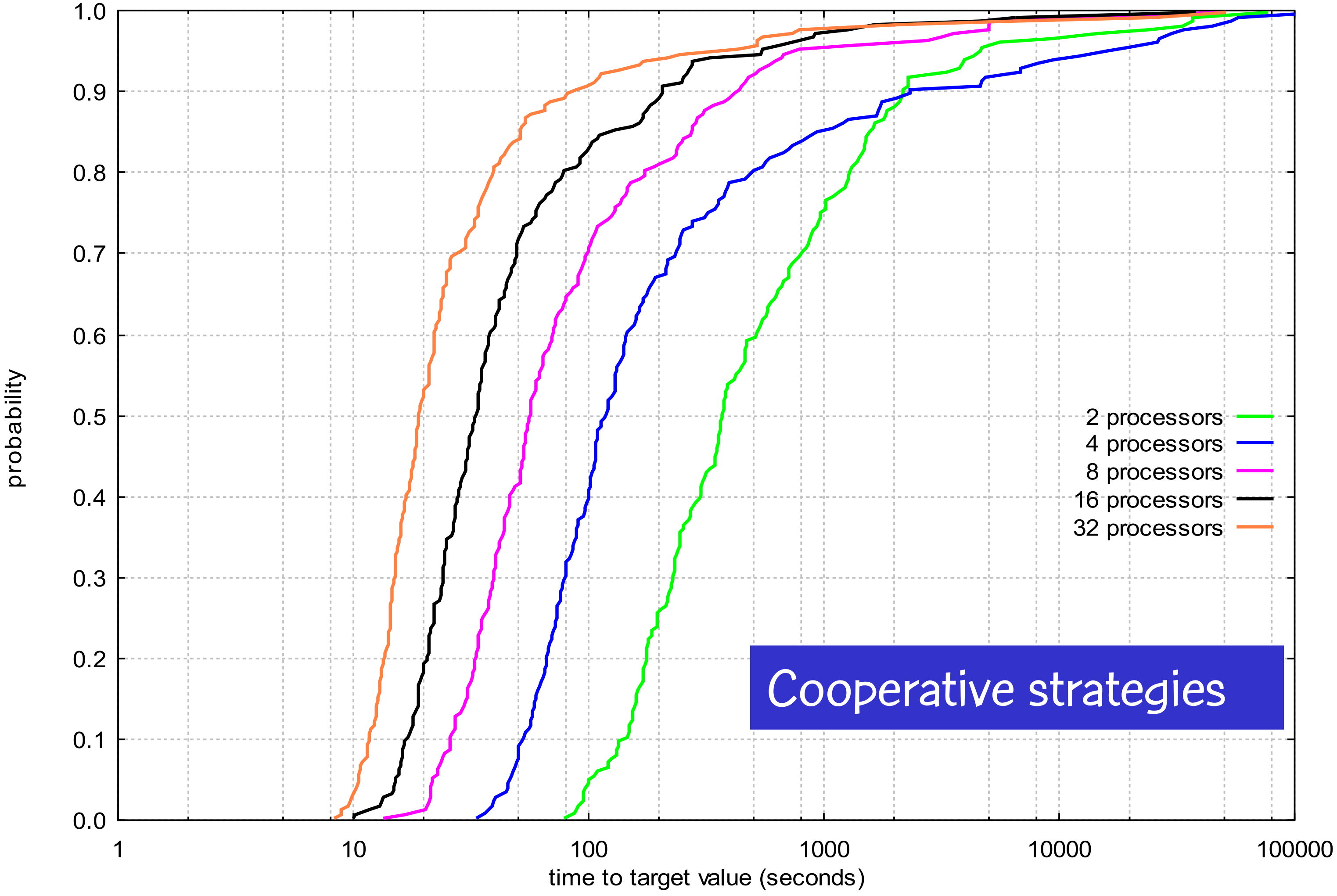
- Linux cluster with 32 Pentium IV 1.7 GHz processors with 256 Mbytes of RAM each
- Extreme Networks switch with 48 10/100 Mbits/s ports and two 1 Gbits/s ports

Parallel environment



Independent strategies

Parallel environment



Cooperative strategies

Remarks

Cooperative parallel strategies based on path-relinking:

– Path-relinking offers a nice strategy to introduce memory and cooperation in parallel implementations.

– Cooperative strategy performs better due to smaller number of iterations and to inter-processor cooperation.

– Linear speedups with the parallel implementation.

– Robustness: cooperative strategy is faster and better.

– Parallel systems are not easily scalable, parallel strategies require careful implementations.

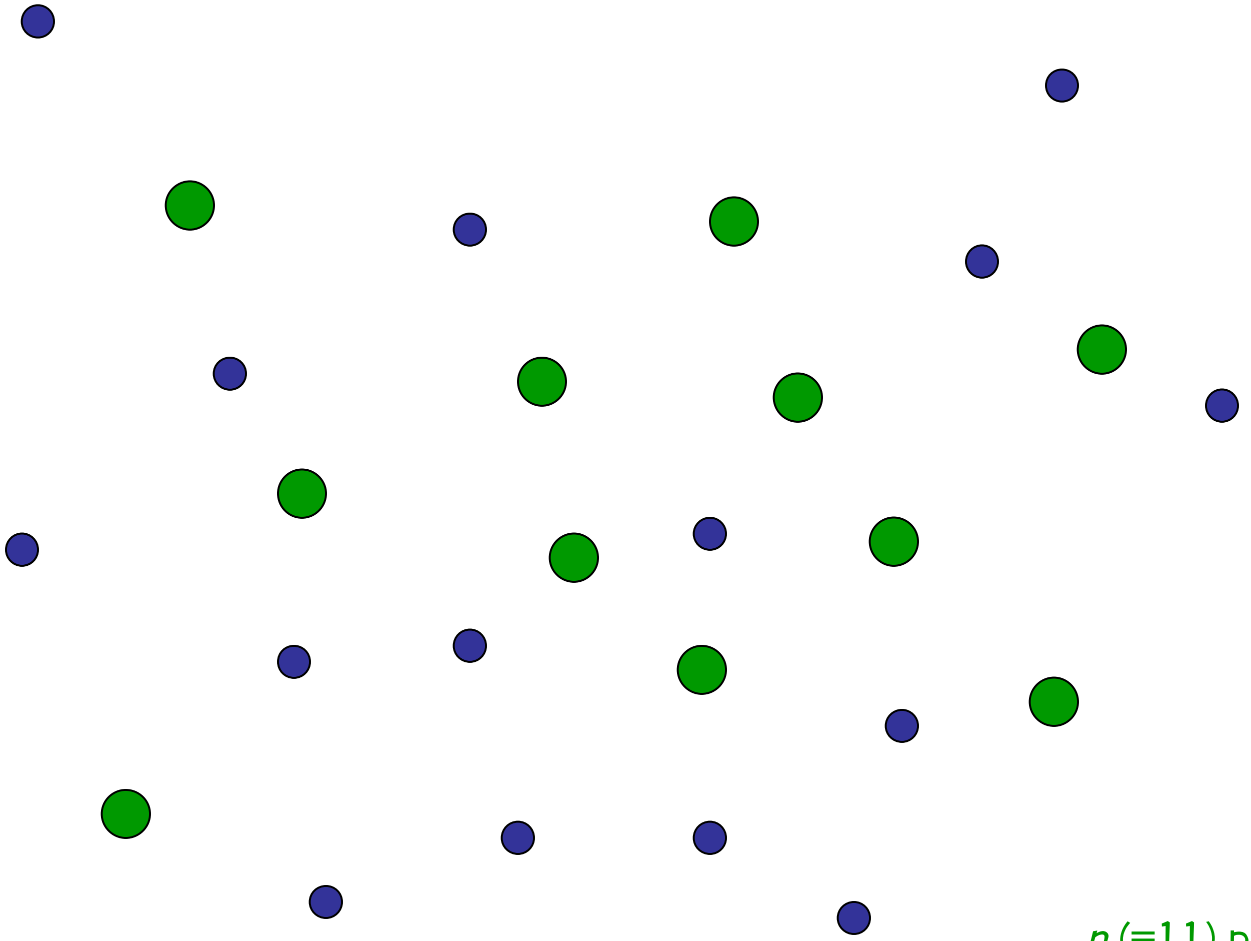
Finding approximate solutions for the p-median problem with GRASP wih EvPR

Joint work with Renato Werneck

Summary

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

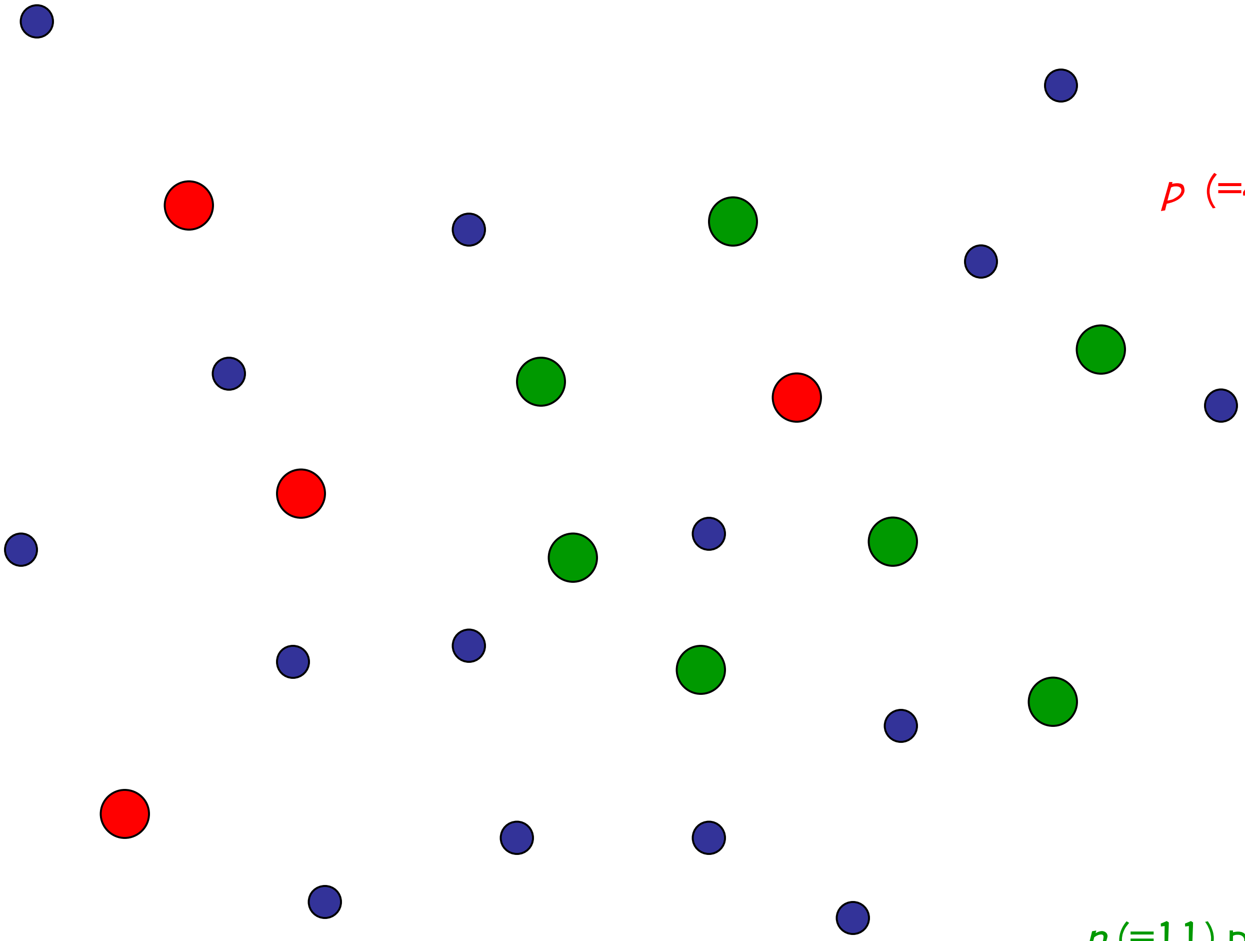
p-median problem



n (=11) potential service locations

m (=15) customers

p-median problem

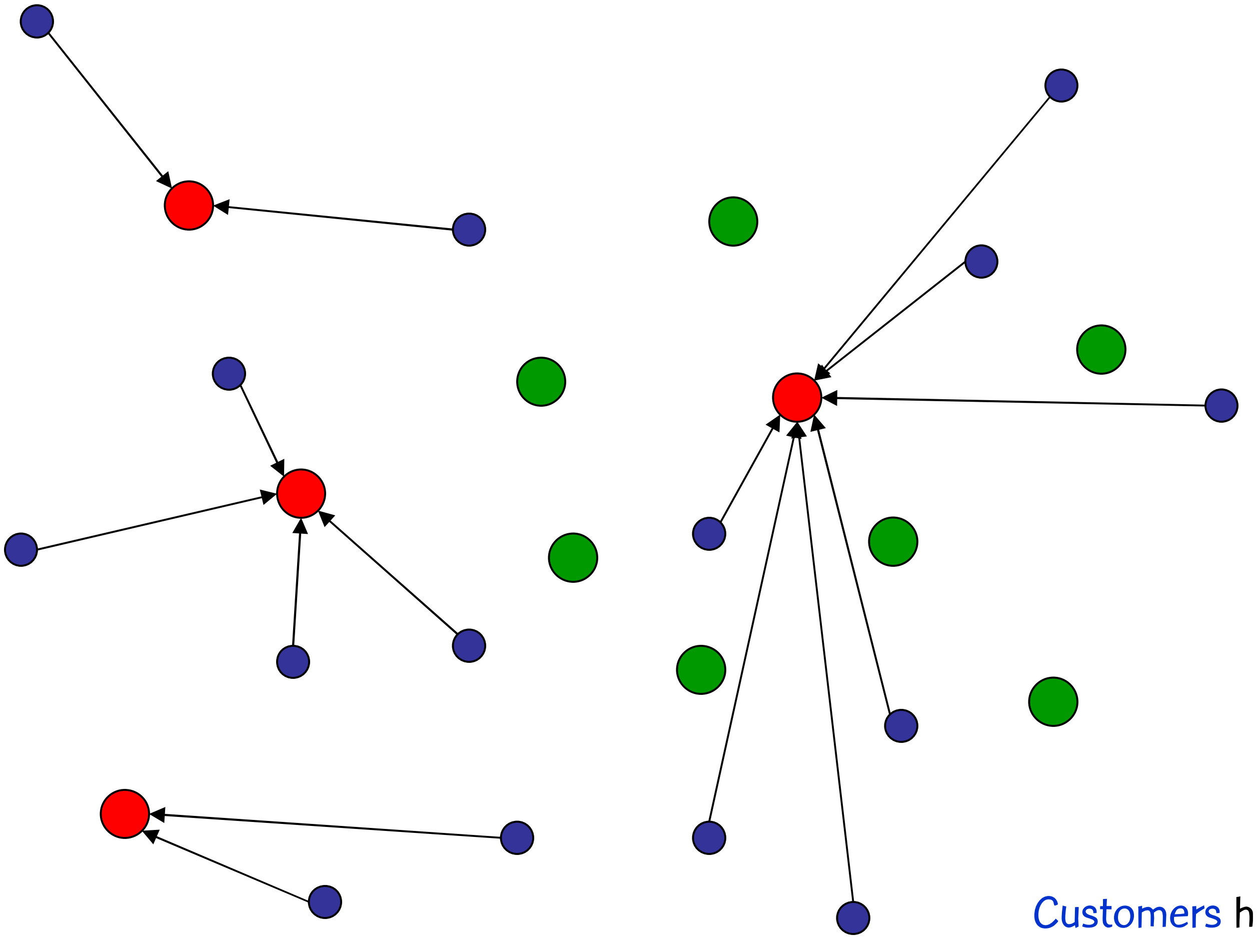


p (=4) service sites to be deployed

n (=11) potential service locations

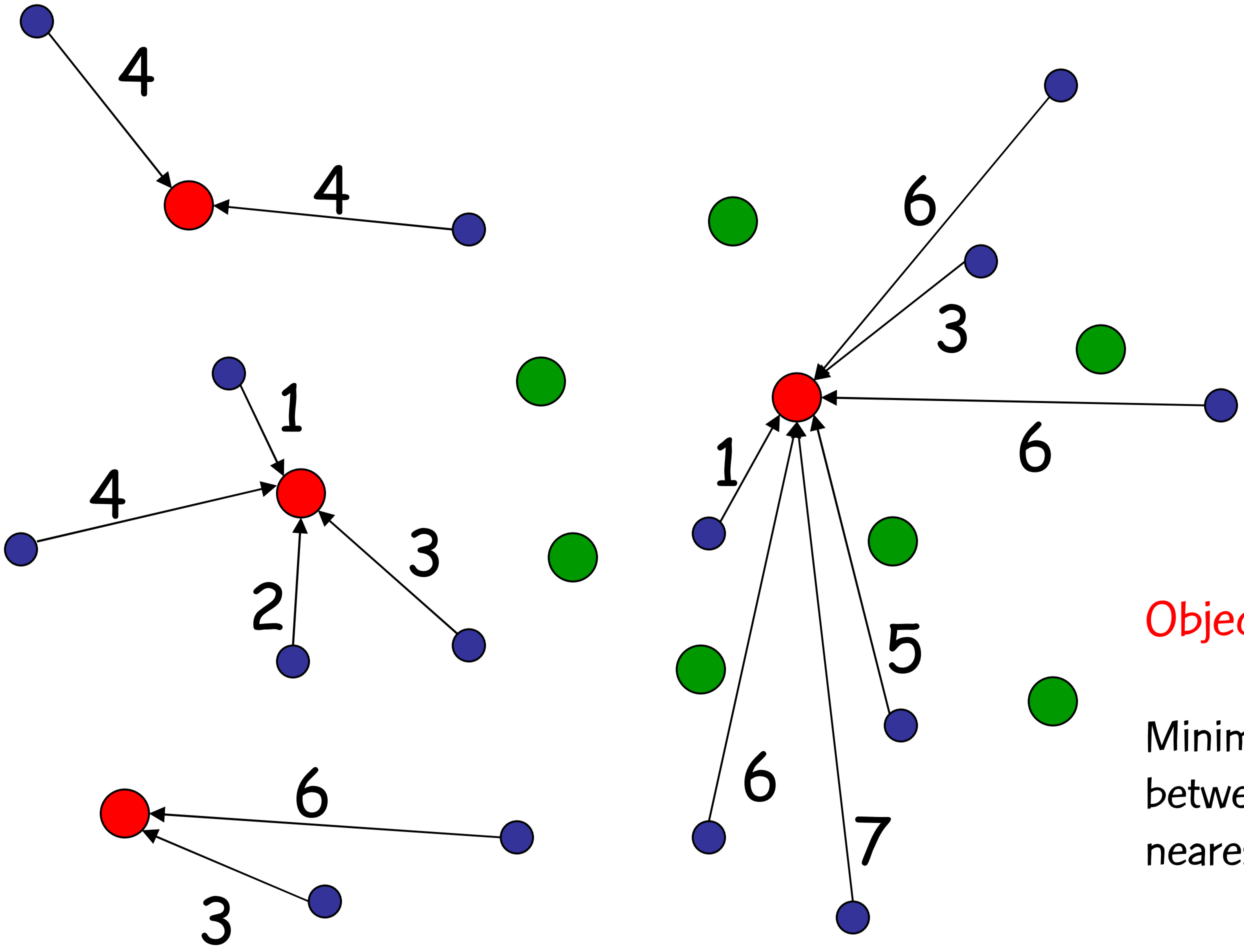
m (=15) customers

p-median problem



Customers home into nearest service center.

p-median problem

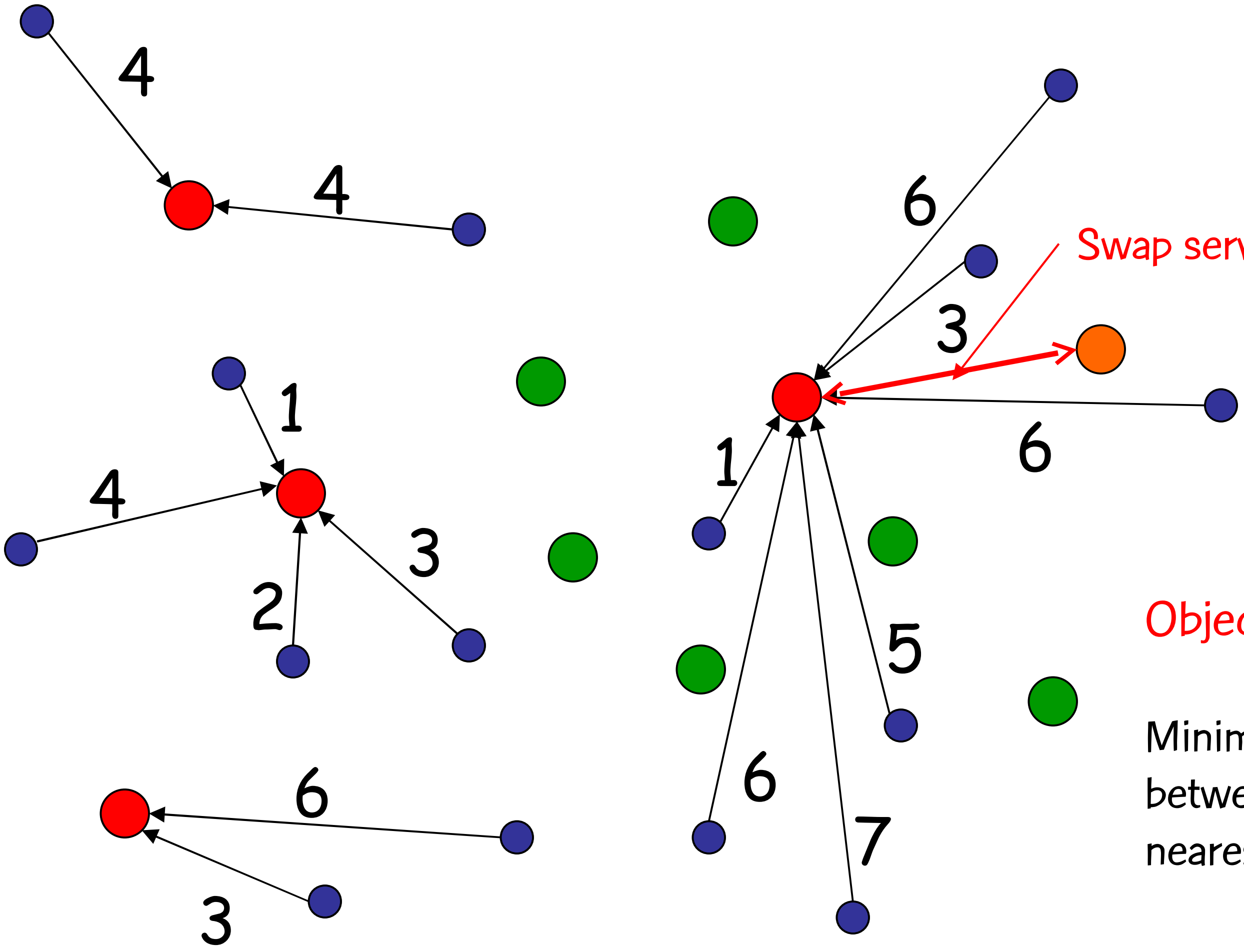


Objective of optimization:

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

Total distance = 61

p-median problem



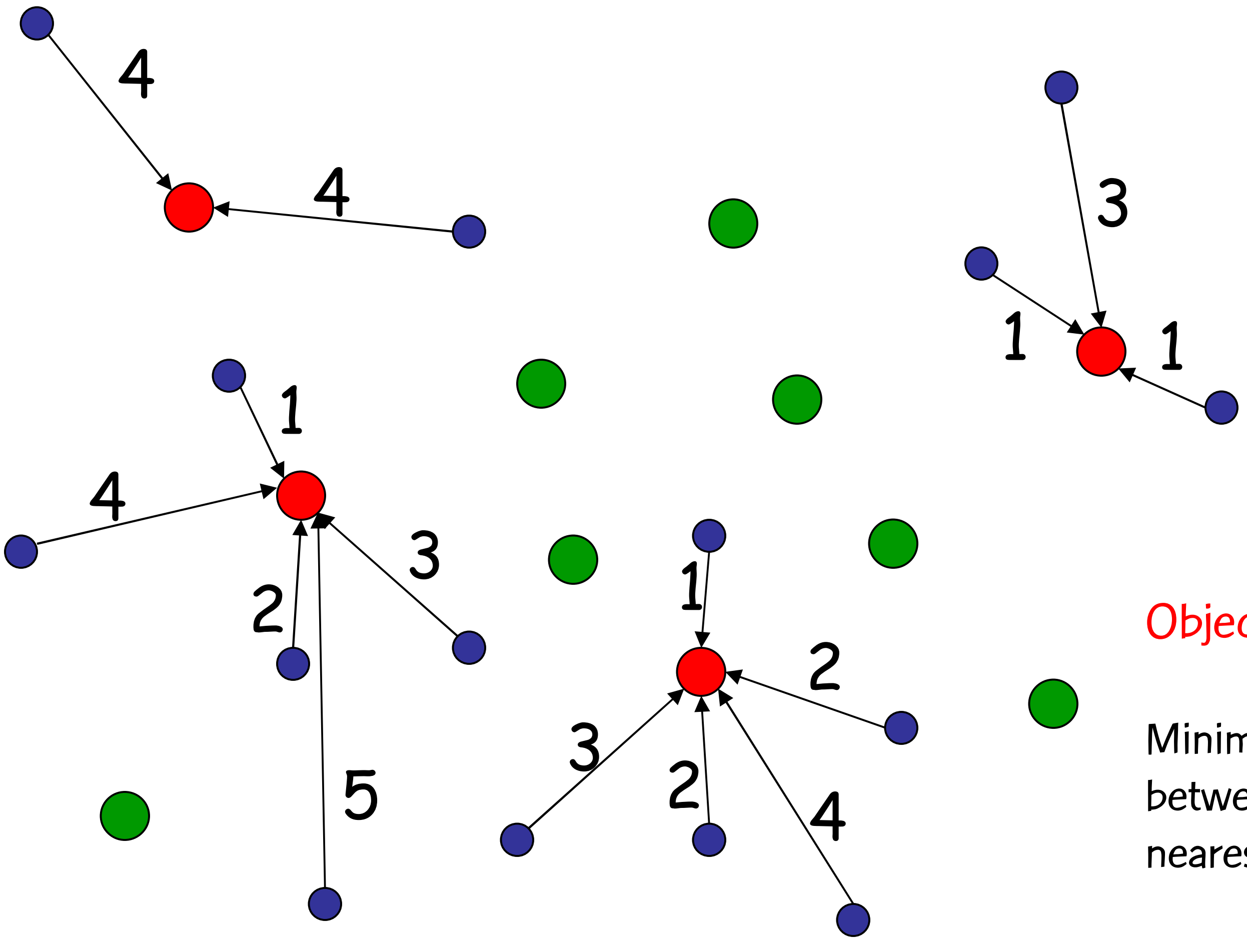
Swap service centers

Objective of optimization:

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

Total distance = 61

p-median problem



Objective of optimization:

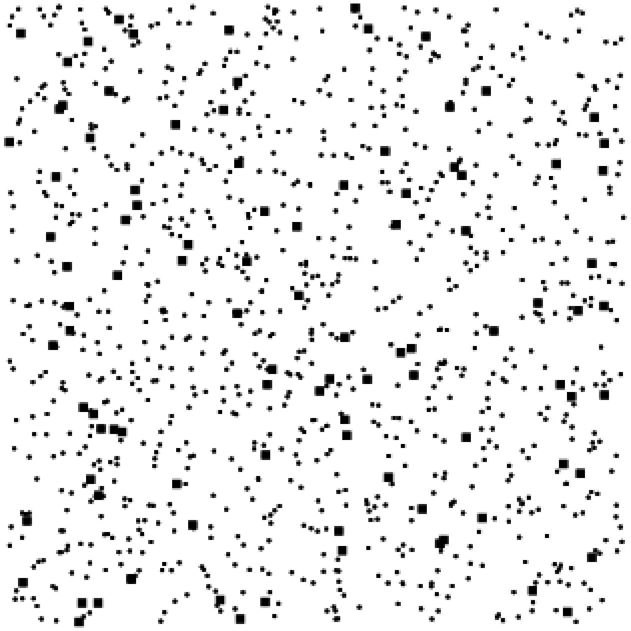
Minimize sum of the distances between customers and their nearest service center.

Total distance = 40 < 61

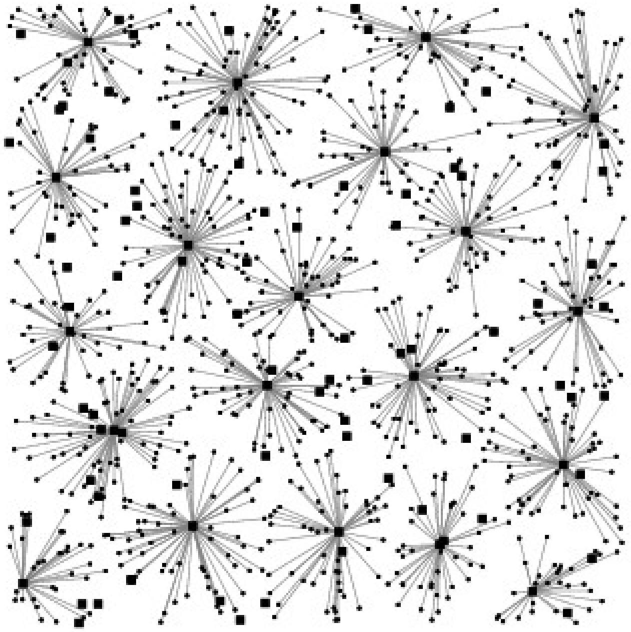
Example: 1000 customer locations, choose best 20 of 100 service locations

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.

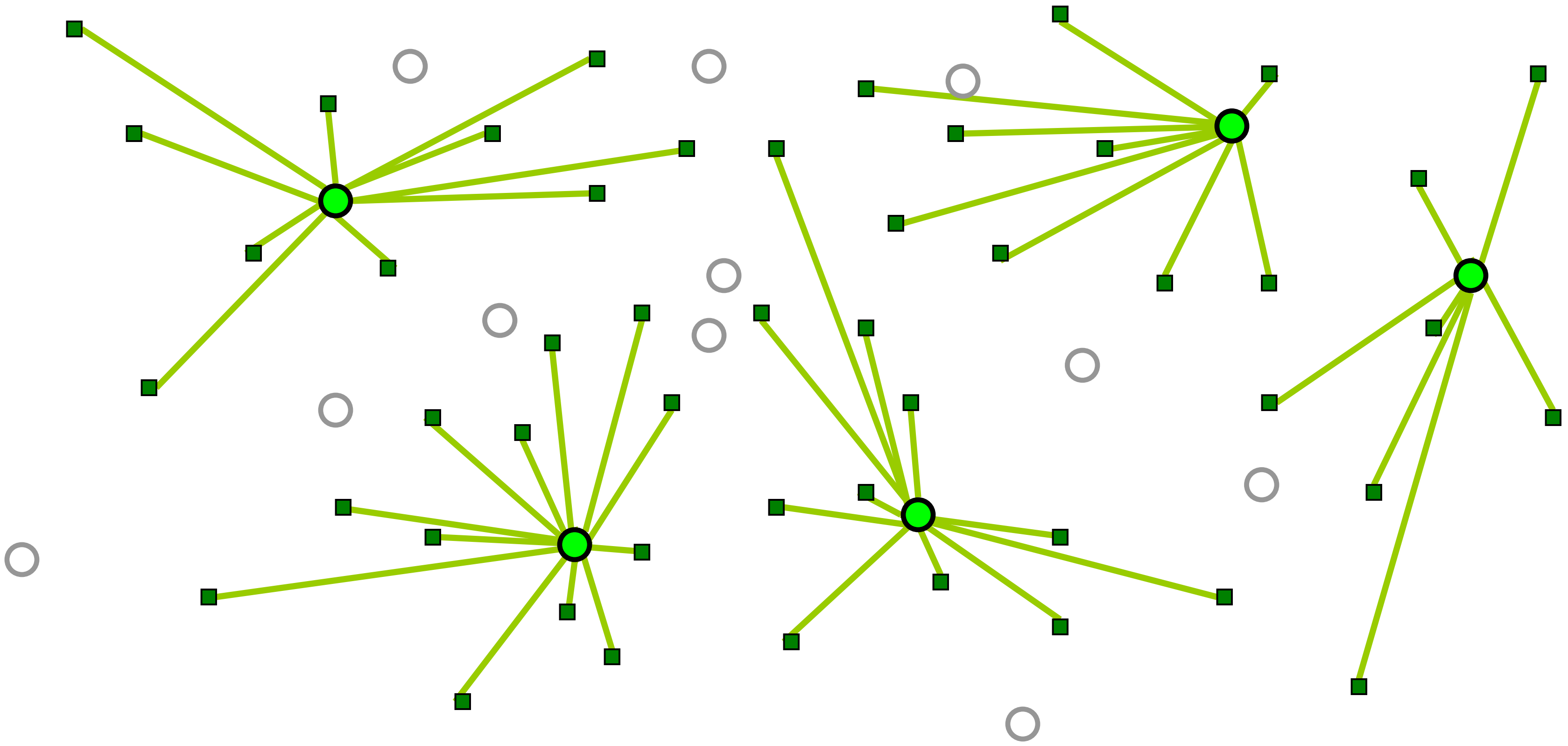
Swap-based local search

Resende & Werneck (Ann. OR, 2007)

Basic Steps:

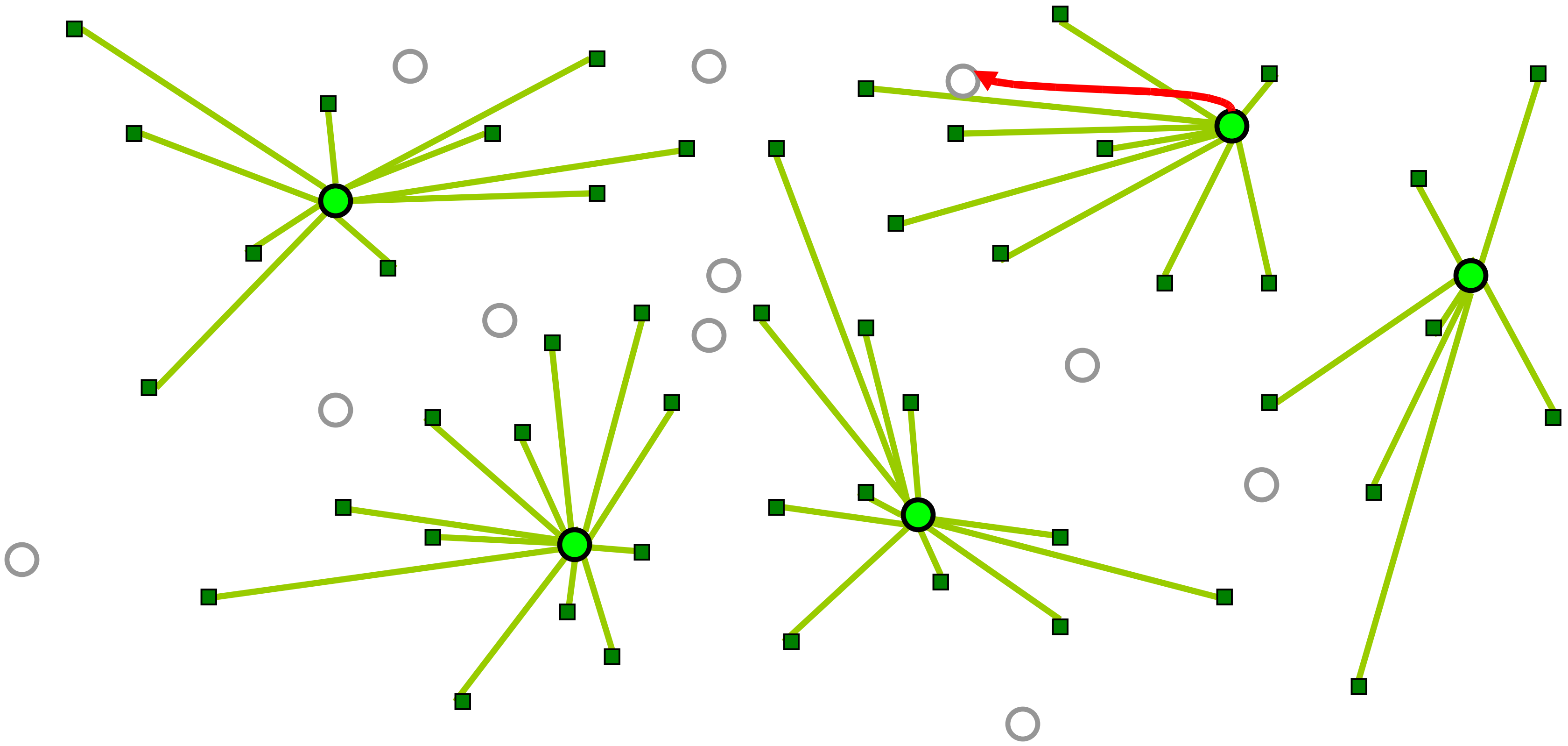
1. Start with some valid solution.
2. Look for a pair of facilities (f_i, f_r) such that:
   - f_i does not belong to the solution;
   - f_r belongs to the solution;
   - swapping f_i and f_r improves the solution.
3. If (2) is successful, swap f_i and f_r and repeat (2); else stop (a local minimum was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

Swap-based local search



original solution

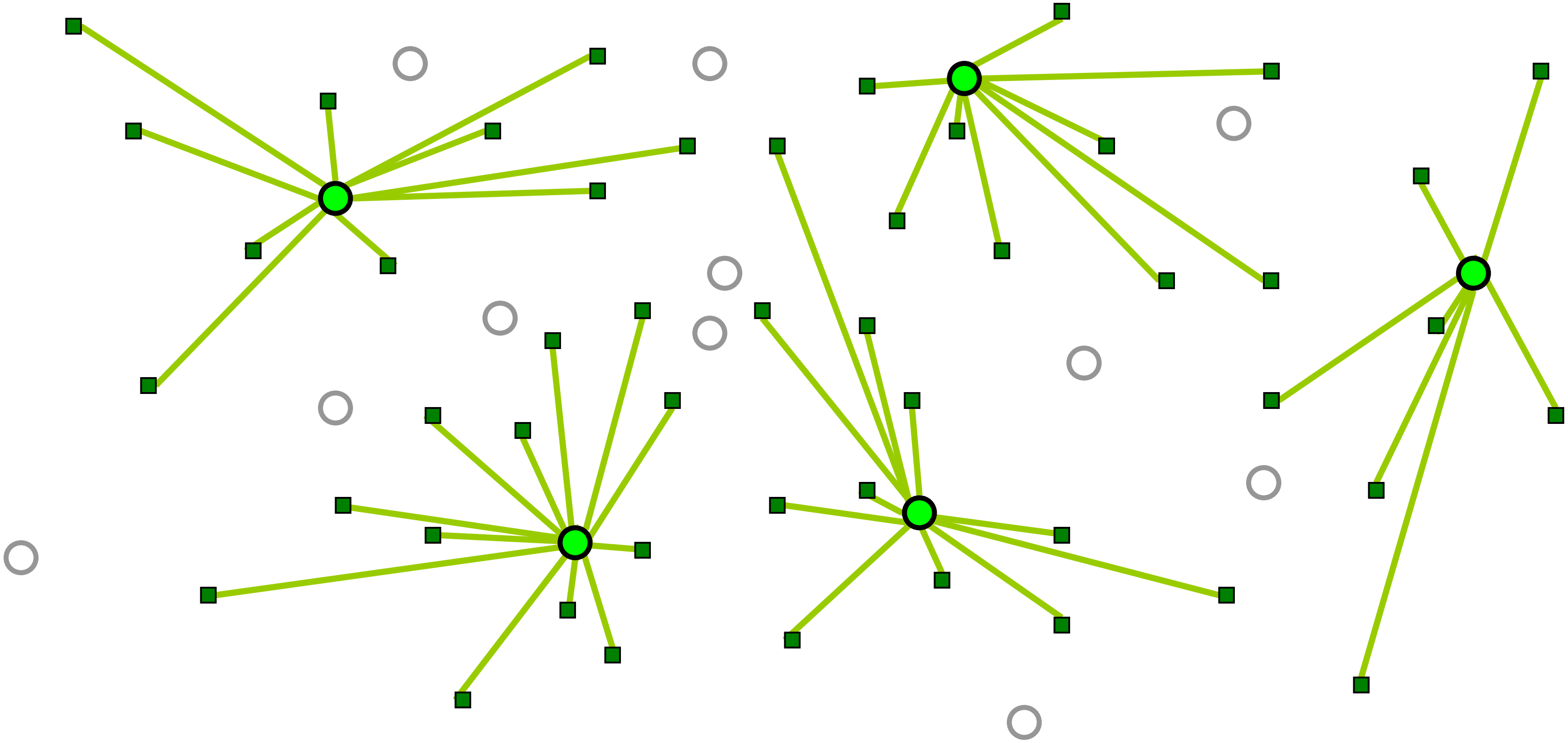
Swap-based local search



original solution

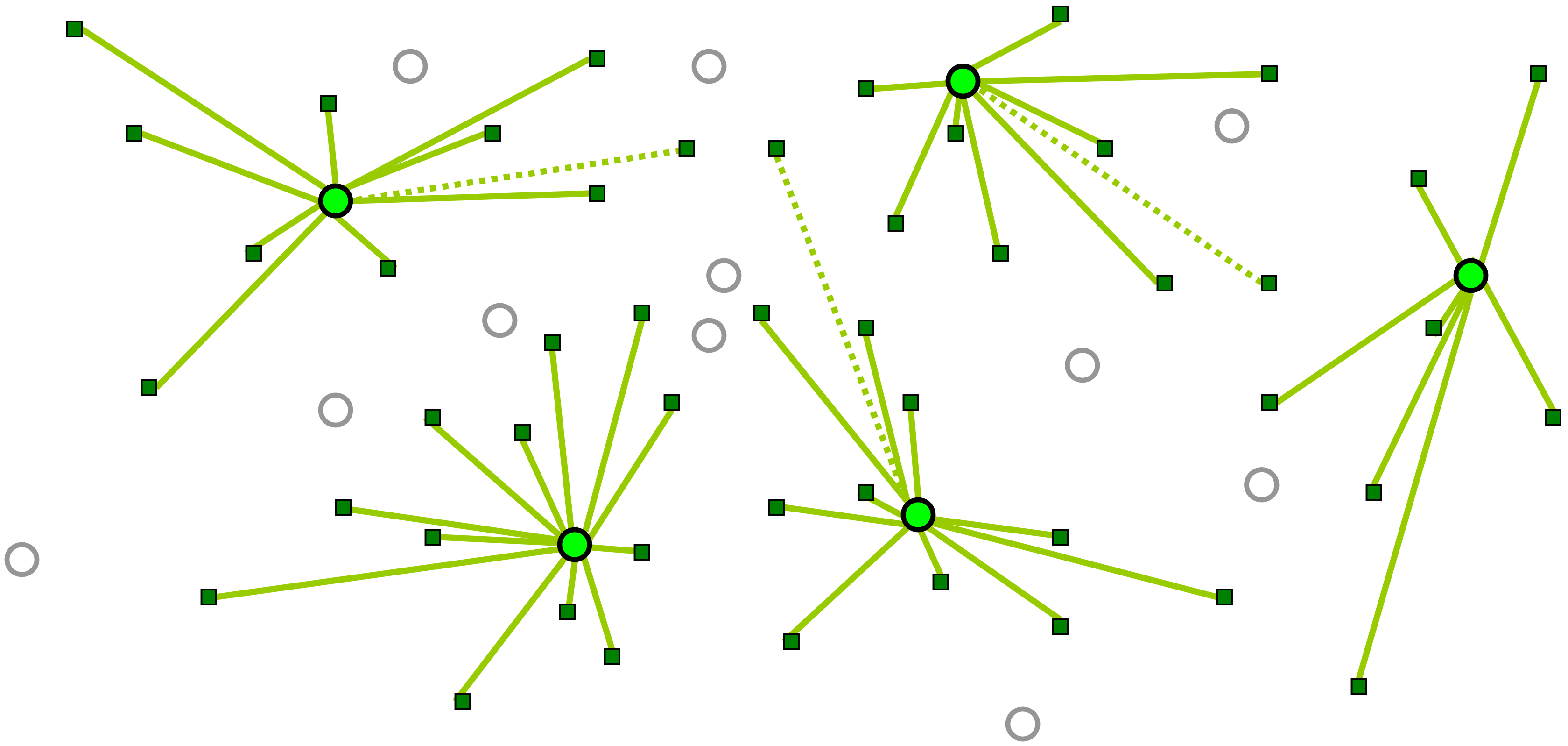
(not a local optimum)

Swap-based local search



improved solution

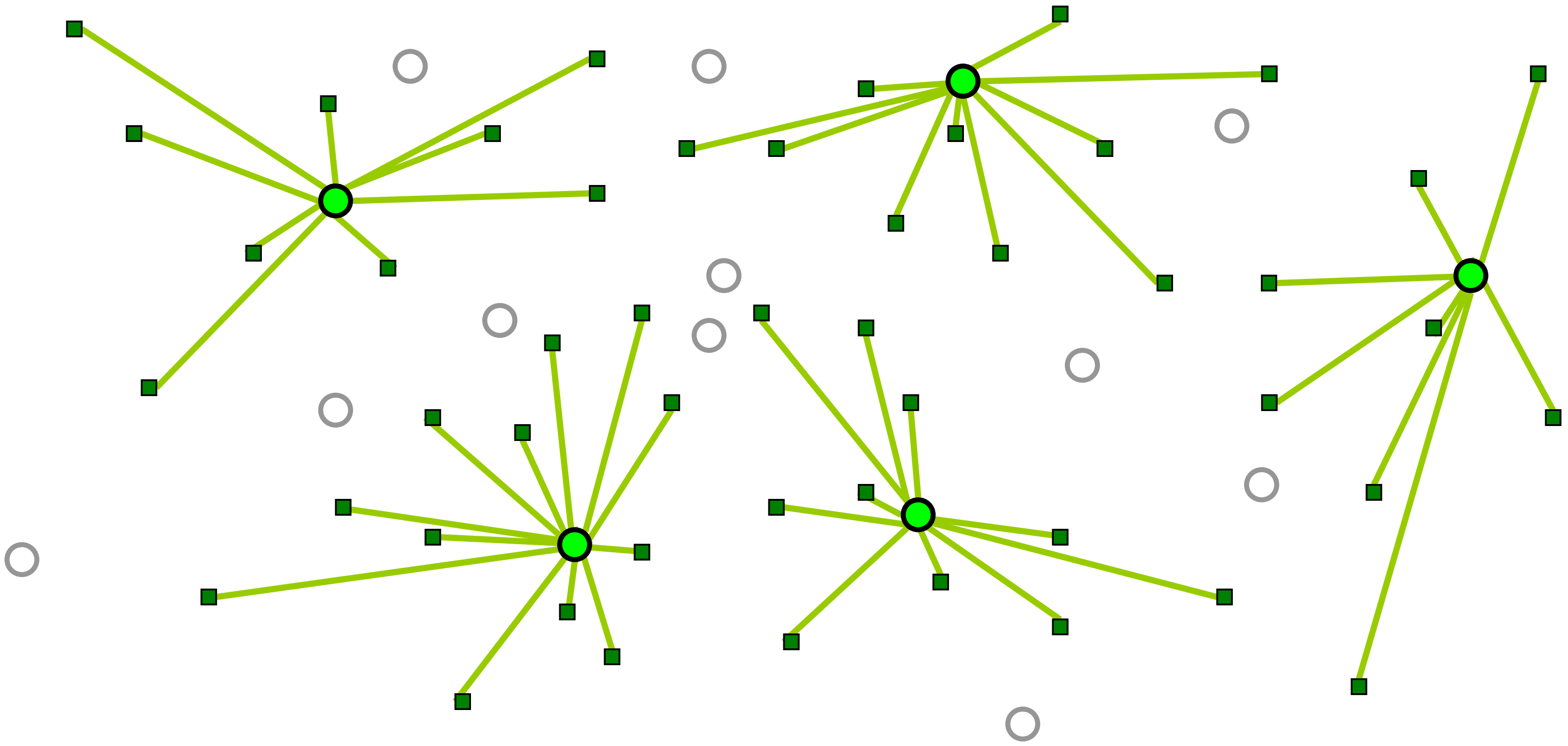
Swap-based local search



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.

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

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

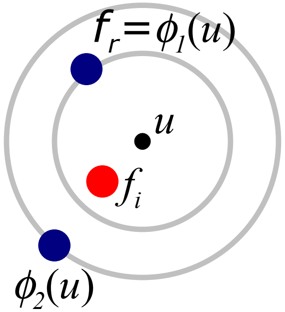
    (for a properly defined extra function).
  - extra can be thought of as a correction factor.

Correction factor

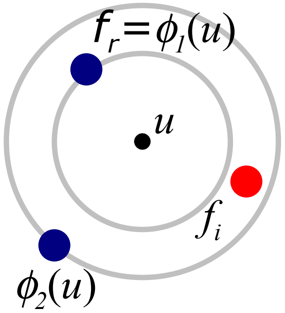
Things will go wrong for a user u iff:

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 ⇒ 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 ($S,\phi_1,\phi_2$)
   $A$ := $U$;
   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;
      $A$ := $\varnothing$;
      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
```

Our implementation

```
function localSearch ($S,\phi_1,\phi_2$)
    $A$ := $U$;
    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
```

Input: solution to be changed and related closeness information.

Our implementation

```
function localSearch ($S, \phi_1, \phi_2$)
    $\boxed{A := U;}$
    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;
        $A := \varnothing$;
        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

```
function localSearch ($S,\phi_1,\phi_2$)
    $A$ := $U$;
    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;
        $A$ := $\varnothing$;
        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
```

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

Our implementation

```
function localSearch ($S,\phi_1,\phi_2$)
  $A$ := $U$;
  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
```

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

Our implementation

```
function localSearch ($S,\phi_1,\phi_2$)
   $A$ := $U$;
   resetStructures($gain,loss,extra$);
   while (TRUE) do {
      forall ($u \in A$) do updateStructures ($S,u,gain,loss,extra,\phi_1,\phi_2$);
      $\boxed{(f_r, f_i, profit) := \text{findBestNeighbor}\ (gain, loss, extra);}$
      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
```

Determine the best swap to make.

Our implementation

```
function localSearch ($S,\phi_1,\phi_2$)
   $A$ := $U$;
   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
```

Swap will be performed only if profitable.

Our implementation

```
function localSearch ($S,\phi_1,\phi_2$)
   $A$ := $U$;
   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;
      $A$ := $\varnothing$;
      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
```

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 ($S,\phi_1,\phi_2$)
    $A$ := $U$;
    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;
        $A$ := $\varnothing$;
        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
```

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

Our implementation

```
function localSearch ($S,\phi_1,\phi_2$)
    $A$ := $U$;
    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 ($profit \le 0$) then break;
        $A$ := $\varnothing$;
        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
```

Finally, perform the swap.

Our implementation

```
function localSearch ($S,\phi_1,\phi_2$)
   $A$ := $U$;
   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;
      $A$ := $\varnothing$;
      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
```

Update closeness information for next iteration.

Bottlenecks

```
function localSearch ($S, \phi_1, \phi_2$)
    $A$ := $U$;
    resetStructures($gain, loss, extra$);
    while (TRUE) do {
3      forall ($u \in A$) do updateStructures ($S, u, gain, loss, extra, \phi_1, \phi_2$);
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
3      forall ($u \in A$) do undoUpdateStructures($S, u, gain, loss, extra, \phi_1, \phi_2$);
       insert($S, f_i$);
       remove($S, f_r$);
1      updateClosest($S, f_i, f_r, \phi_1, \phi_2$);
    endwhile
end localSearch
```

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

Bottleneck 1: Closeness

```
function localSearch ($S, \phi_1, \phi_2$)
    $A$ := $U$;
    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;
        $A$ := $\varnothing$;
        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($S, f_r$);
        $\boxed{\text{updateClosest}(S, f_i, f_r, \phi_1, \phi_2);}$
    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).

Bottleneck 2: Best neighbor

```
function localSearch ($S,\phi_1,\phi_2$)
   $A$ := $U$;
   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
```

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 ($S,\phi_1,\phi_2$)
   $A$ := $U$;
   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
```

Bottleneck 3 – Update Structures

```
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
```

This loop always takes *m-p* iterations.

Bottleneck 3 – Update Structures

```
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 ($f_i \notin S$ such that $d(u,f_i)<d(u,\phi_2(u))$) do
        $gain[f_i]$ += $\max\{0, d(u,\phi_1(u)) - d(u,f_i)\}$;
        $extra[f_i,f_r]$ += $d(u,\phi_2(u)) - \max\{d(u,f_i), d(u,f_r)\}$;
    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, ...).

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

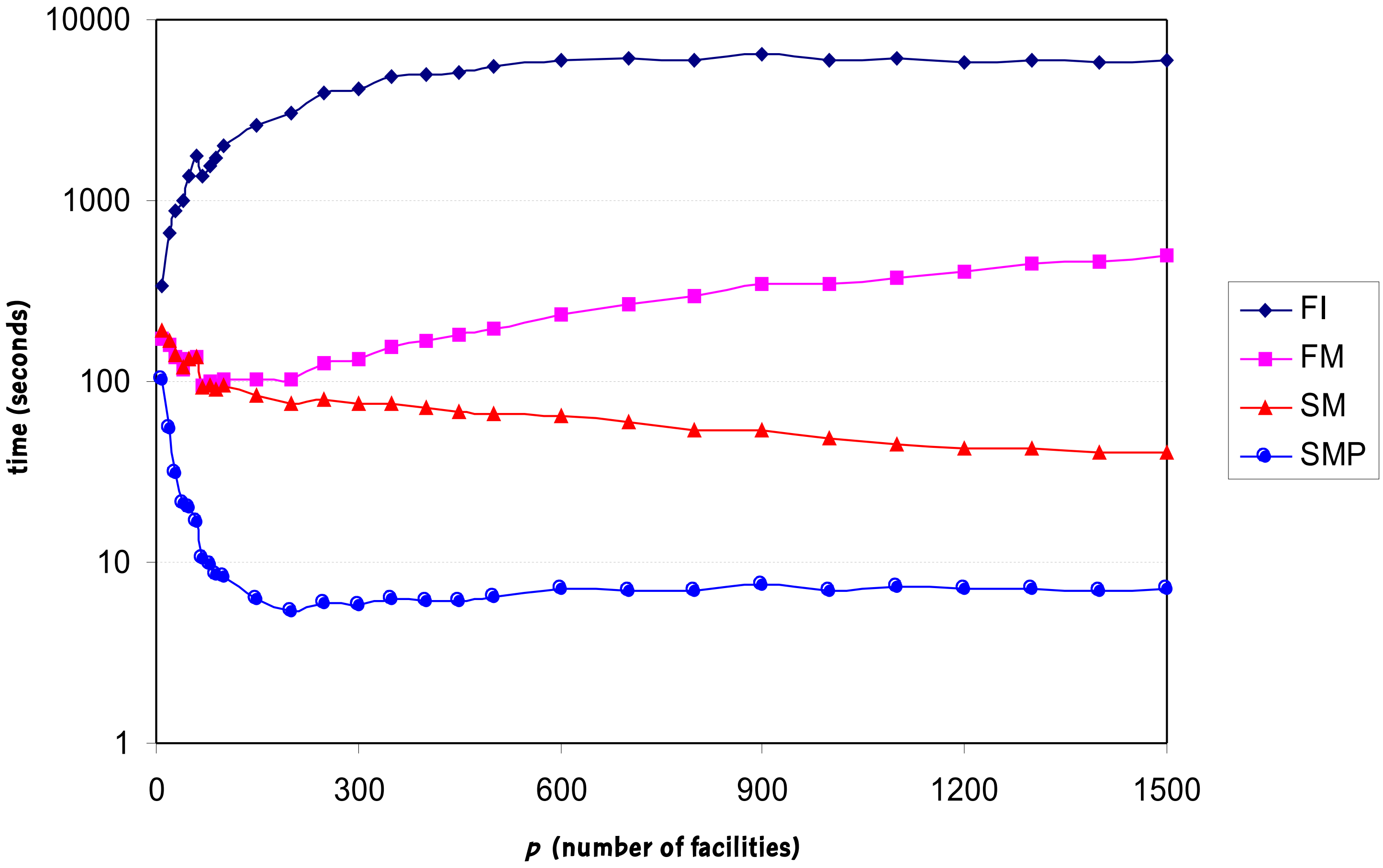
Speedups w.r.t. Whitaker's **FI** (worst cases):

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.

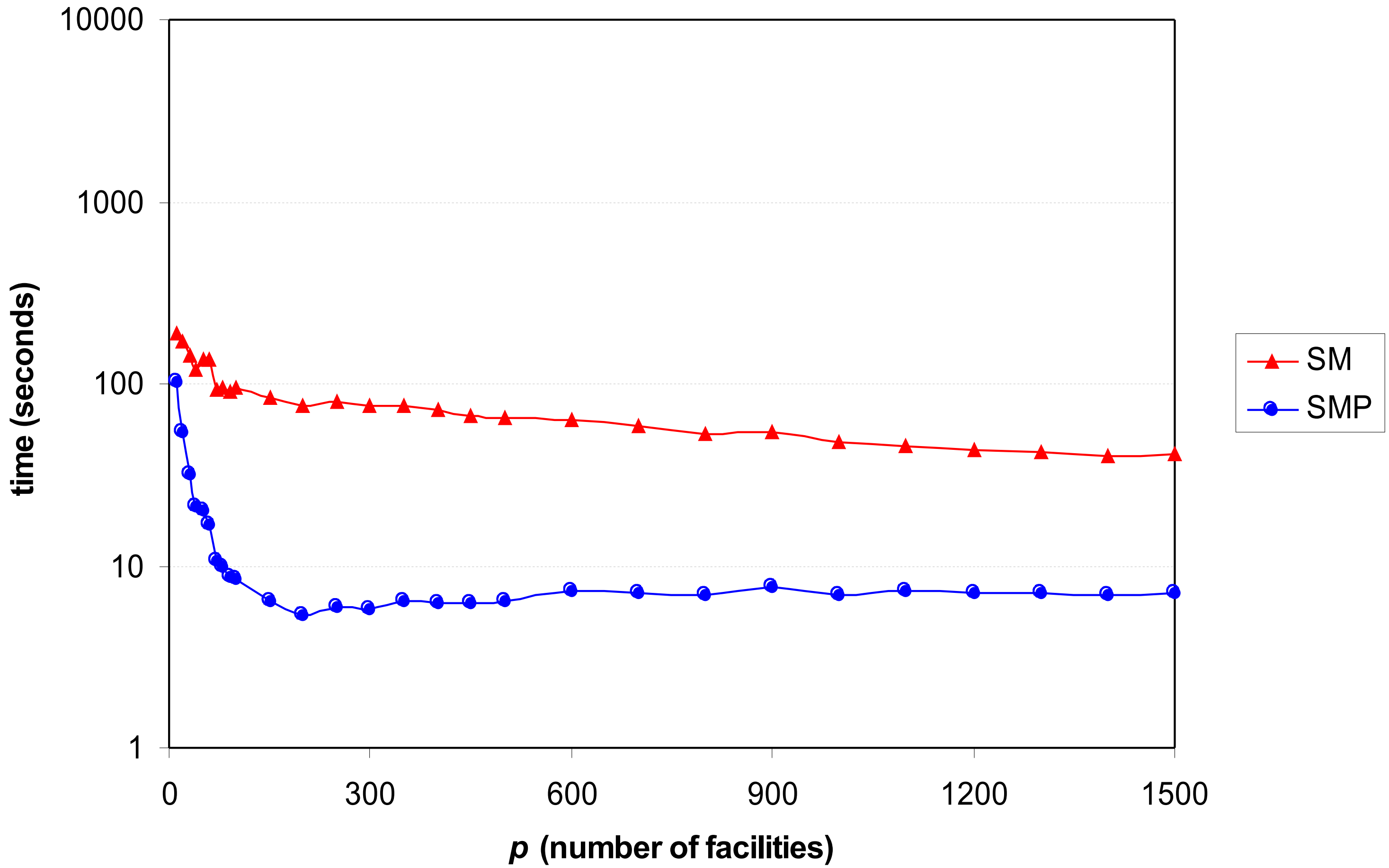
Results



Largest instance tested: 5934 users, Euclidean.

(preprocessing times not considered)

Results

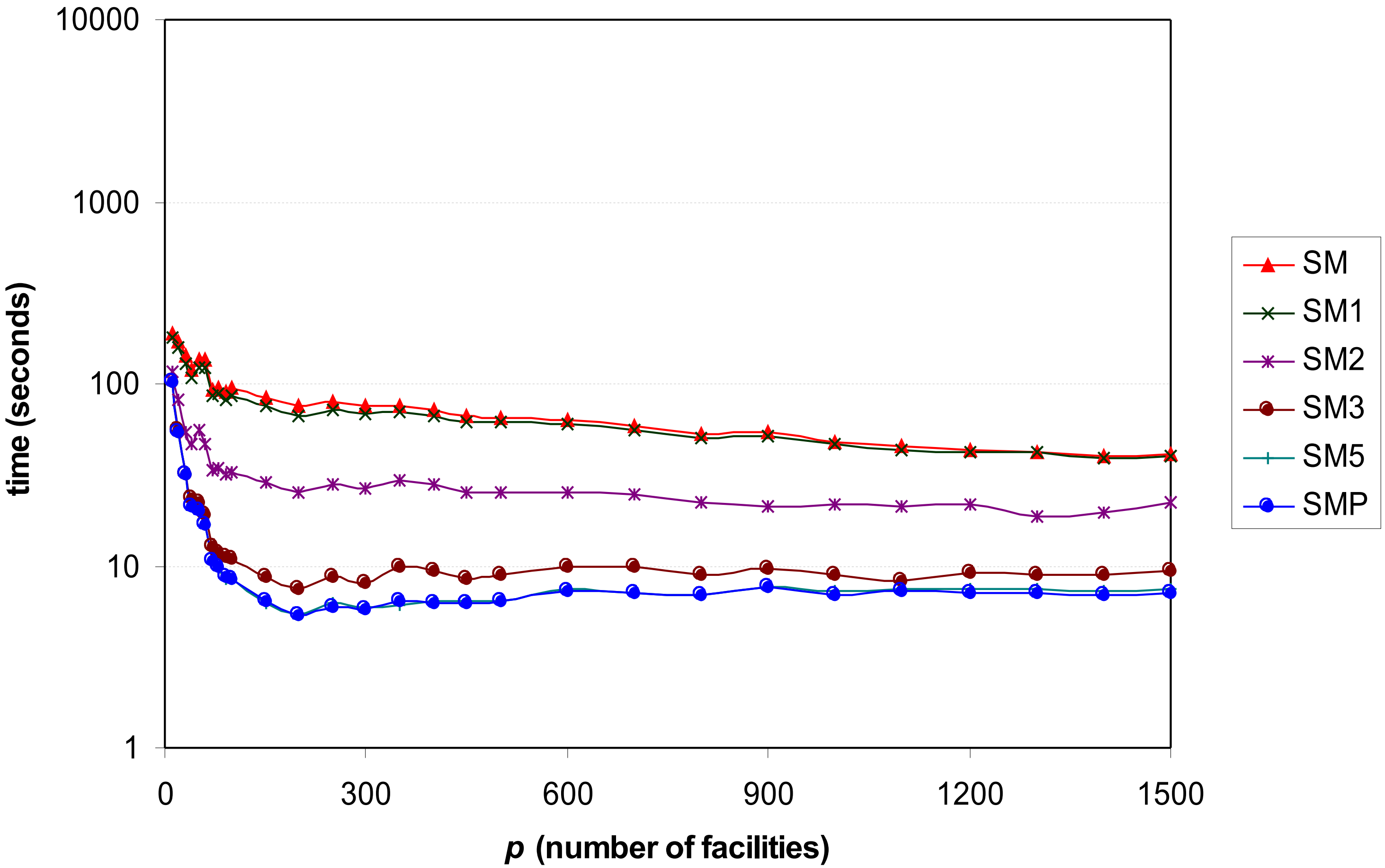


Note that preprocessing significantly accelerates the algorithm.

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

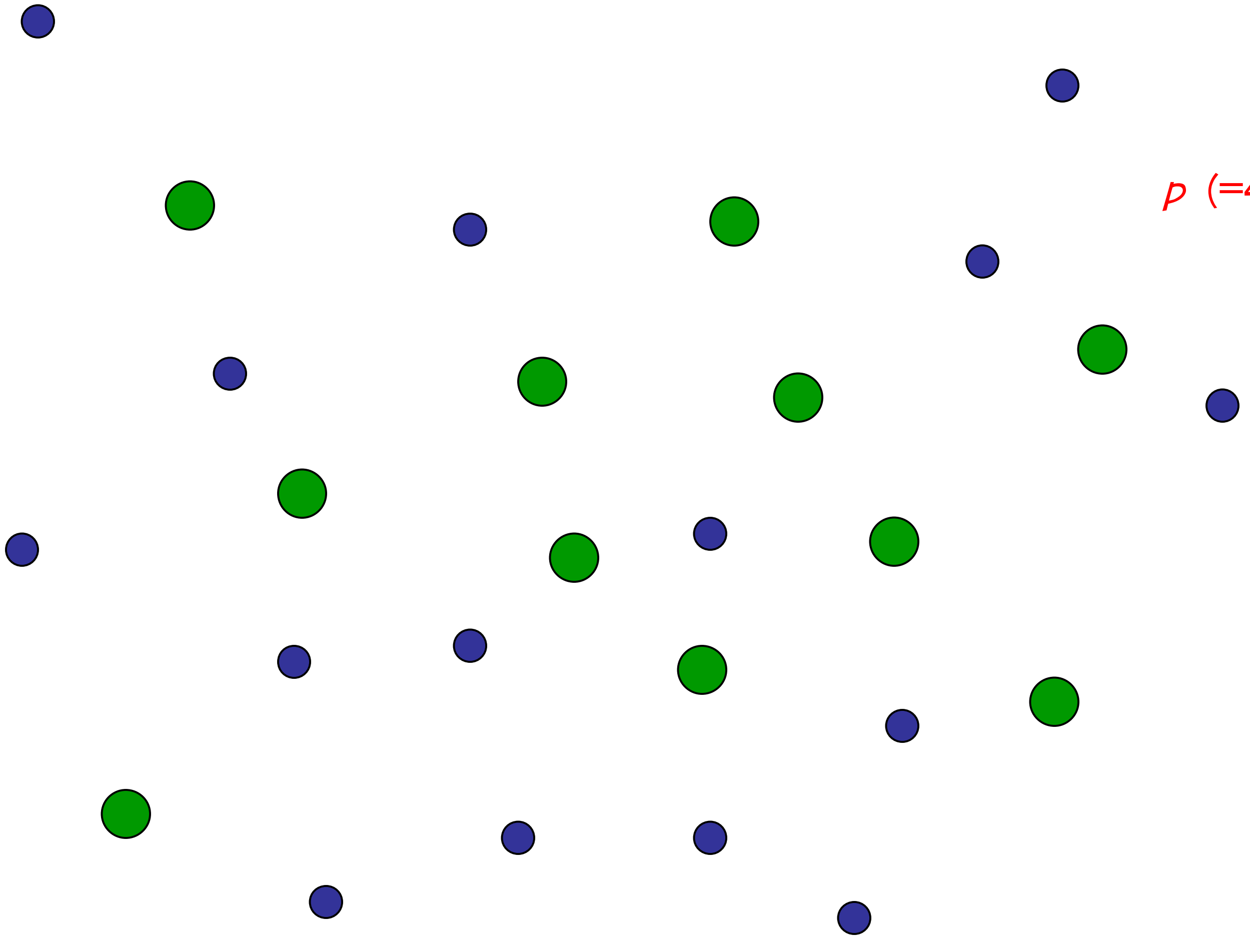
- New implementation of well-known local search.
- Uses extra memory, but much faster in practice.
- Accelerations are metric-independent.
- Especially useful for metaheuristics:
  - We have implemented two GRASPs based on this local search with very promising results.
  - Other existing methods may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

GRASP for p-median

Resende & Werneck (J. Heuristics, 2004)

- A GRASP with evolutionary path-relinking, using the new swap based local search was presented.

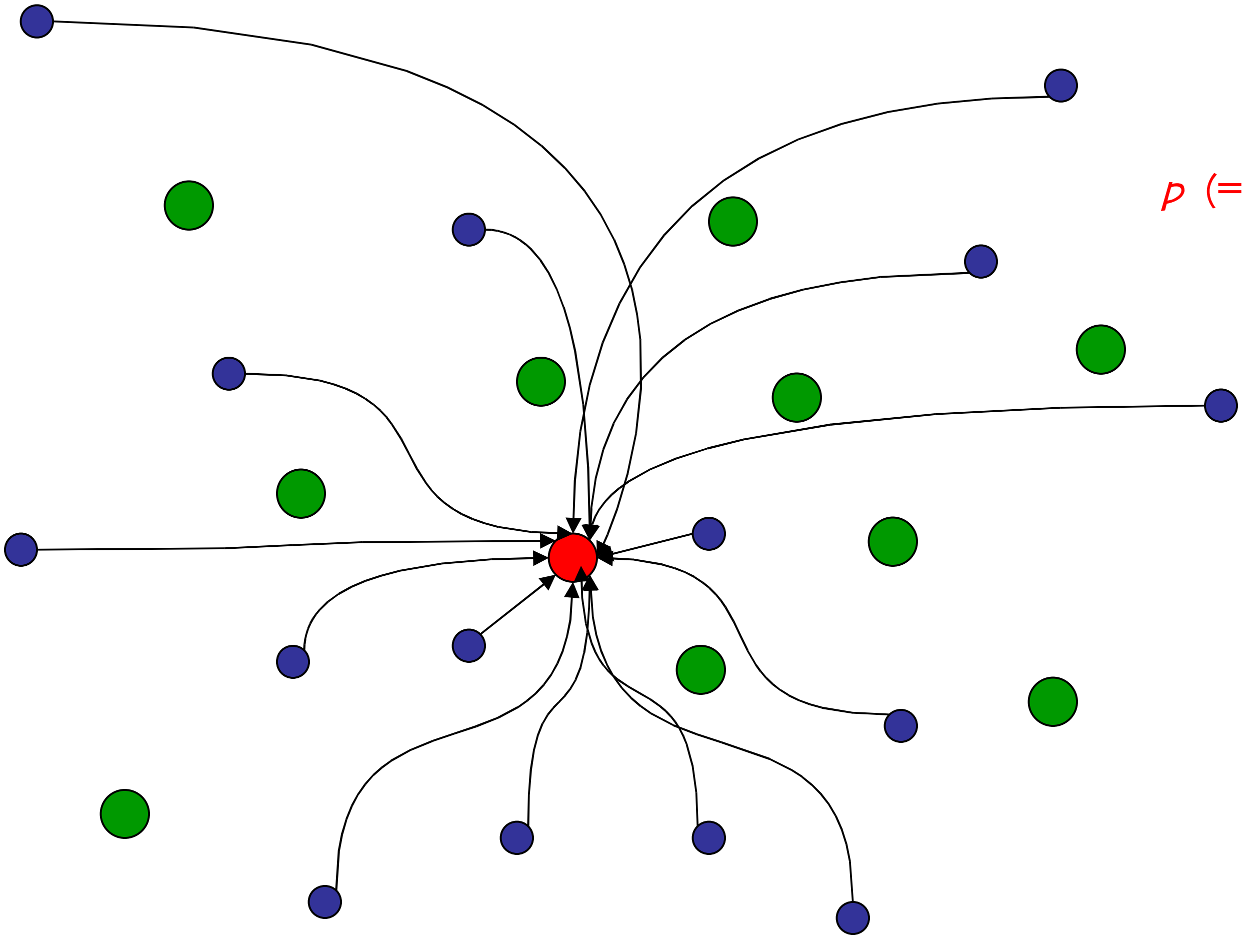
Greedy construction for p-median



p (=4) service sites to be deployed

Start with empty solution

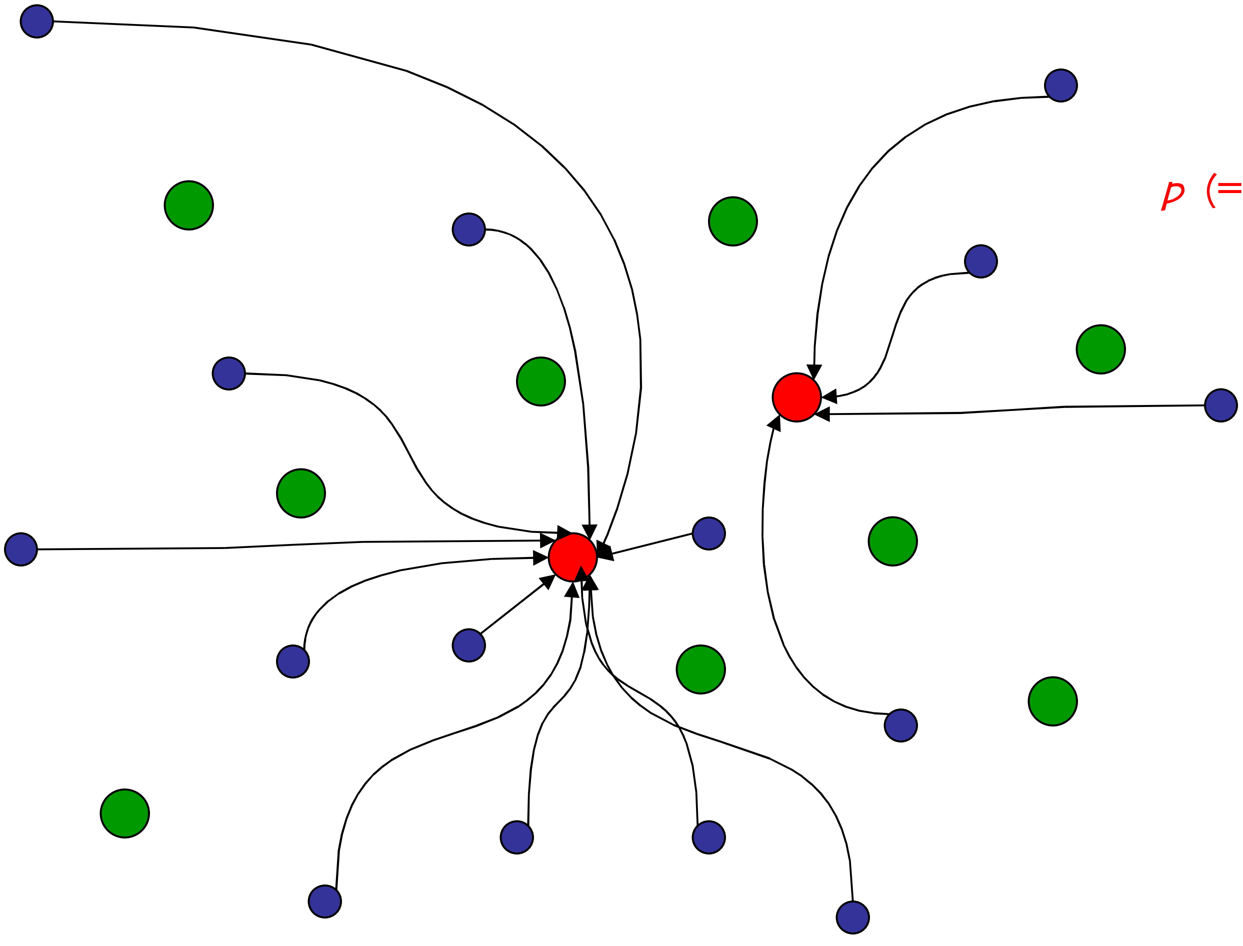
Greedy construction for p-median



p (=4) service sites to be deployed

Add most profitable center (least cost)

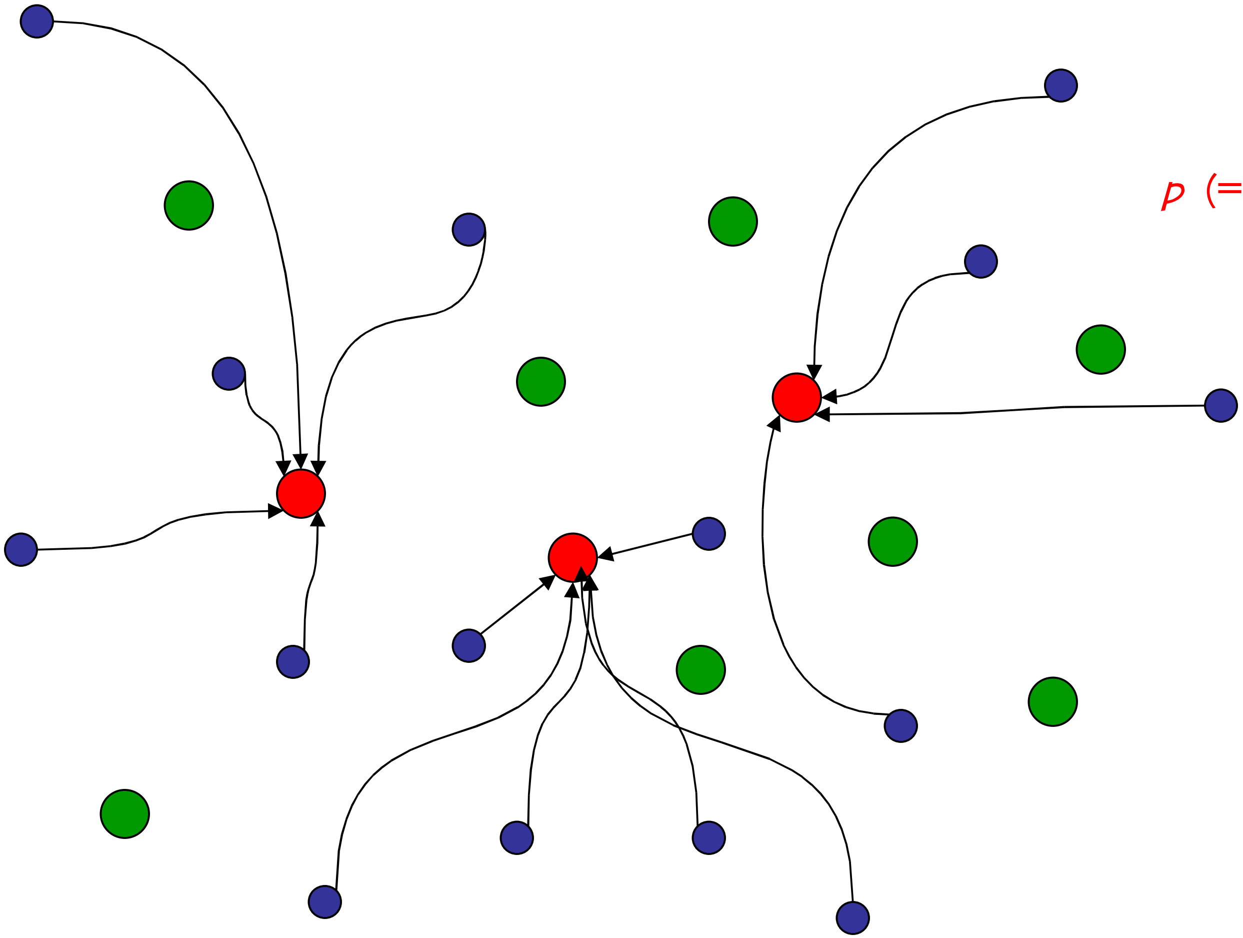
Greedy construction for p-median



p (=4) service sites to be deployed

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

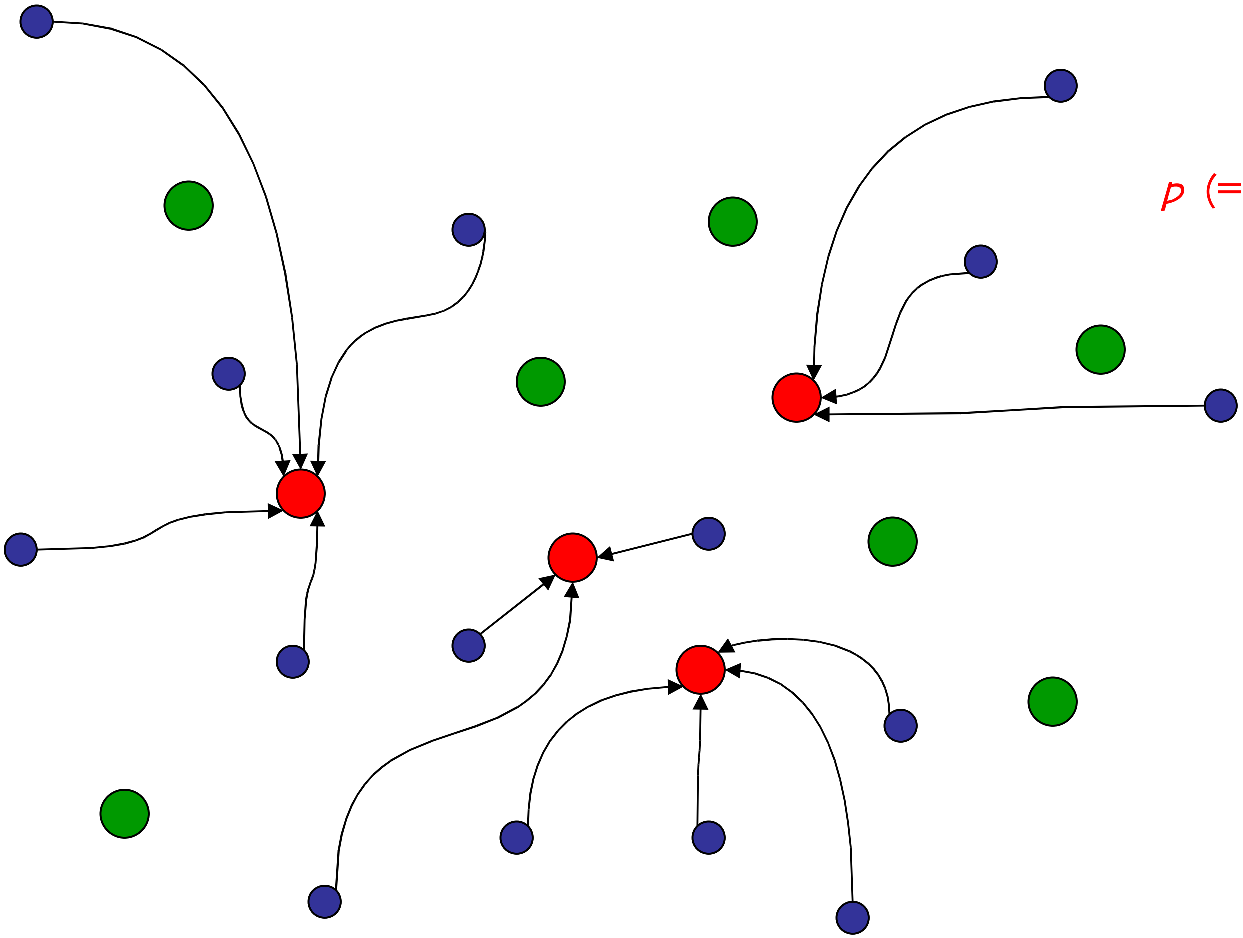
Greedy construction for p-median



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.

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, ... 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*.

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.

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.

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.

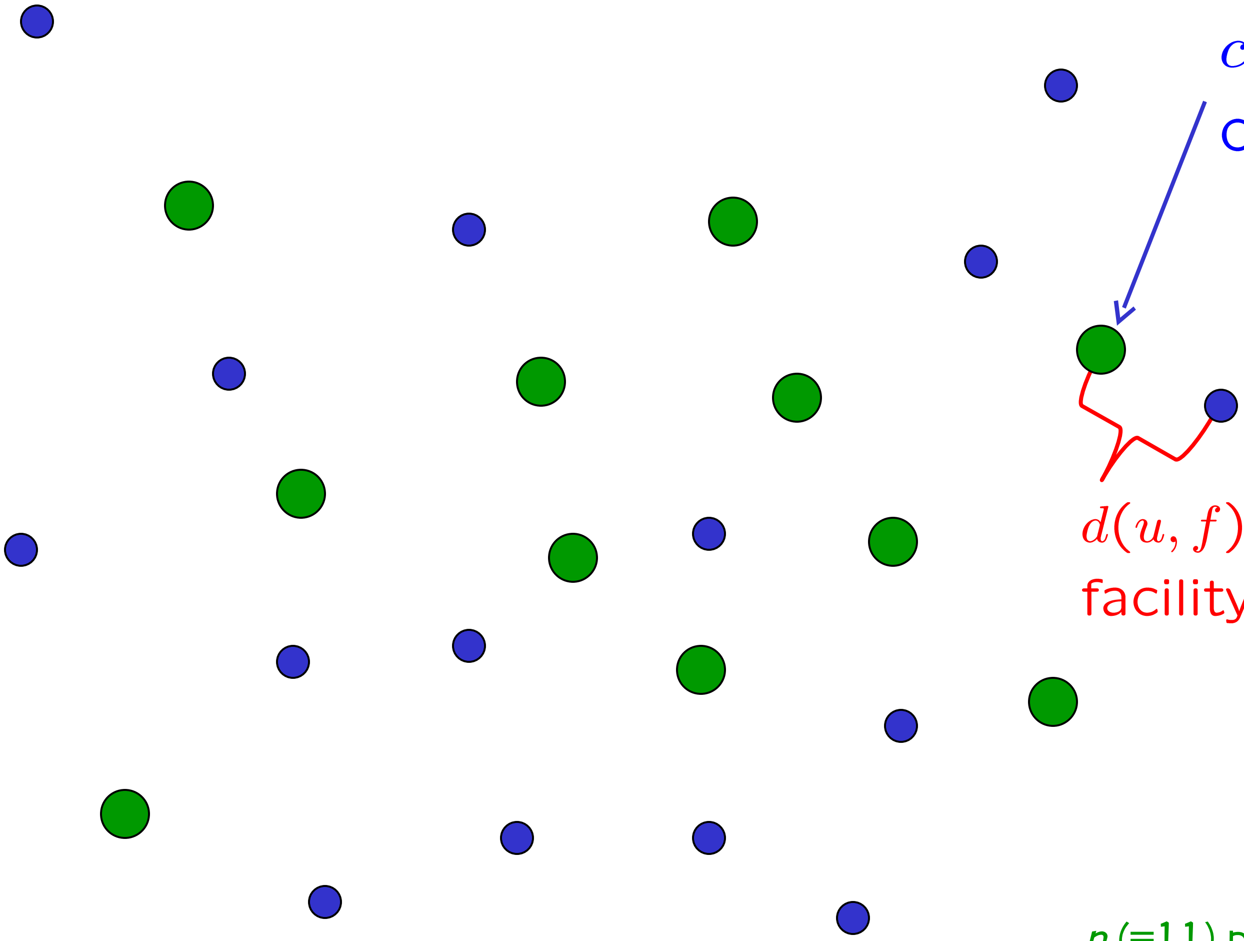
Remarks

- 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.

GRASP with EvPR for Uncapacitated Facility Location

Resende & Werneck (EJOR, 2007)

Uncapacitated facility location problem



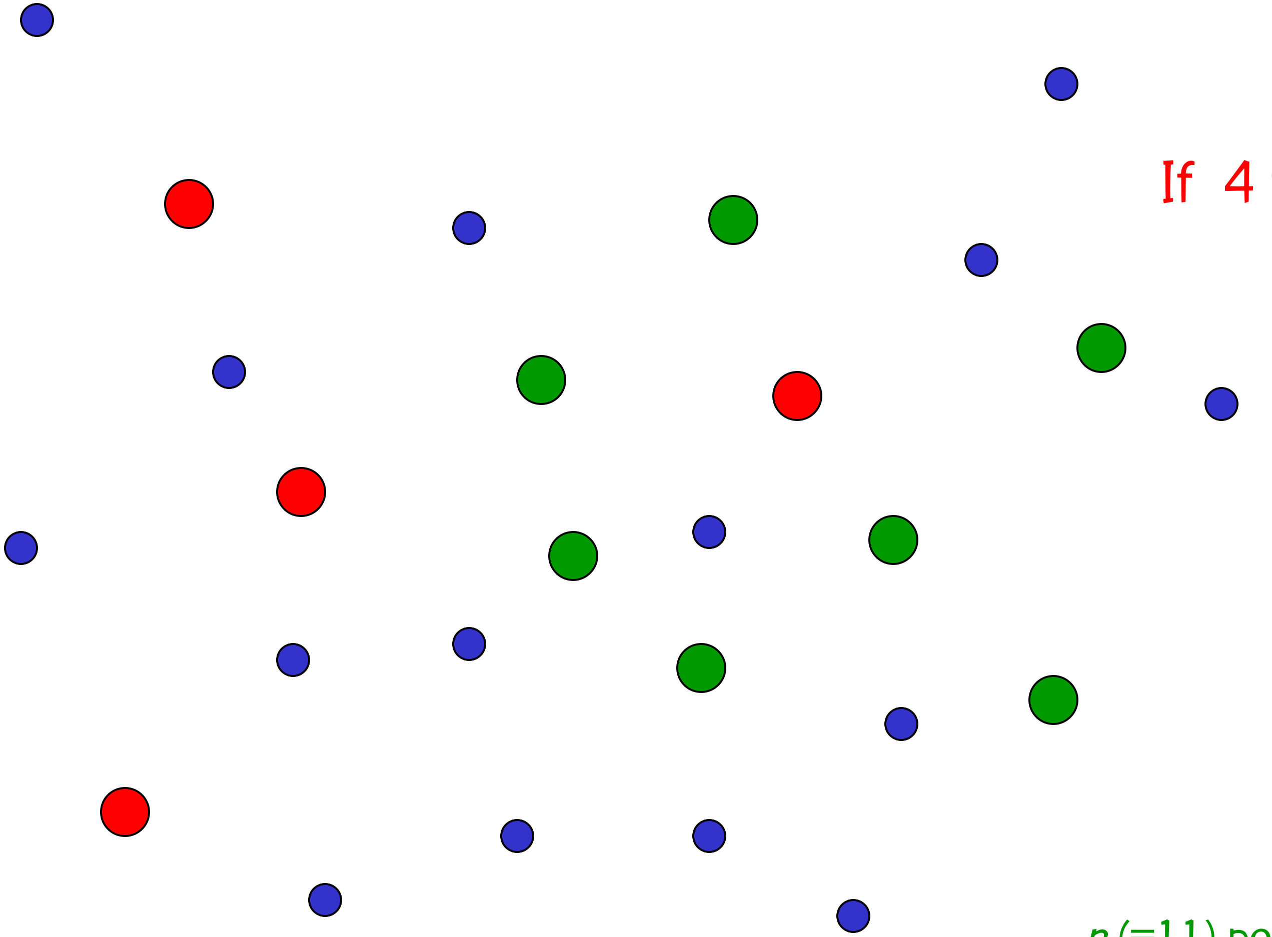
$c(f)$: setup cost of opening facility f

$d(u,f)$: distance between facility d and user u.

n (=11) potential facility locations

m (=15) customers

Uncapacitated facility location problem

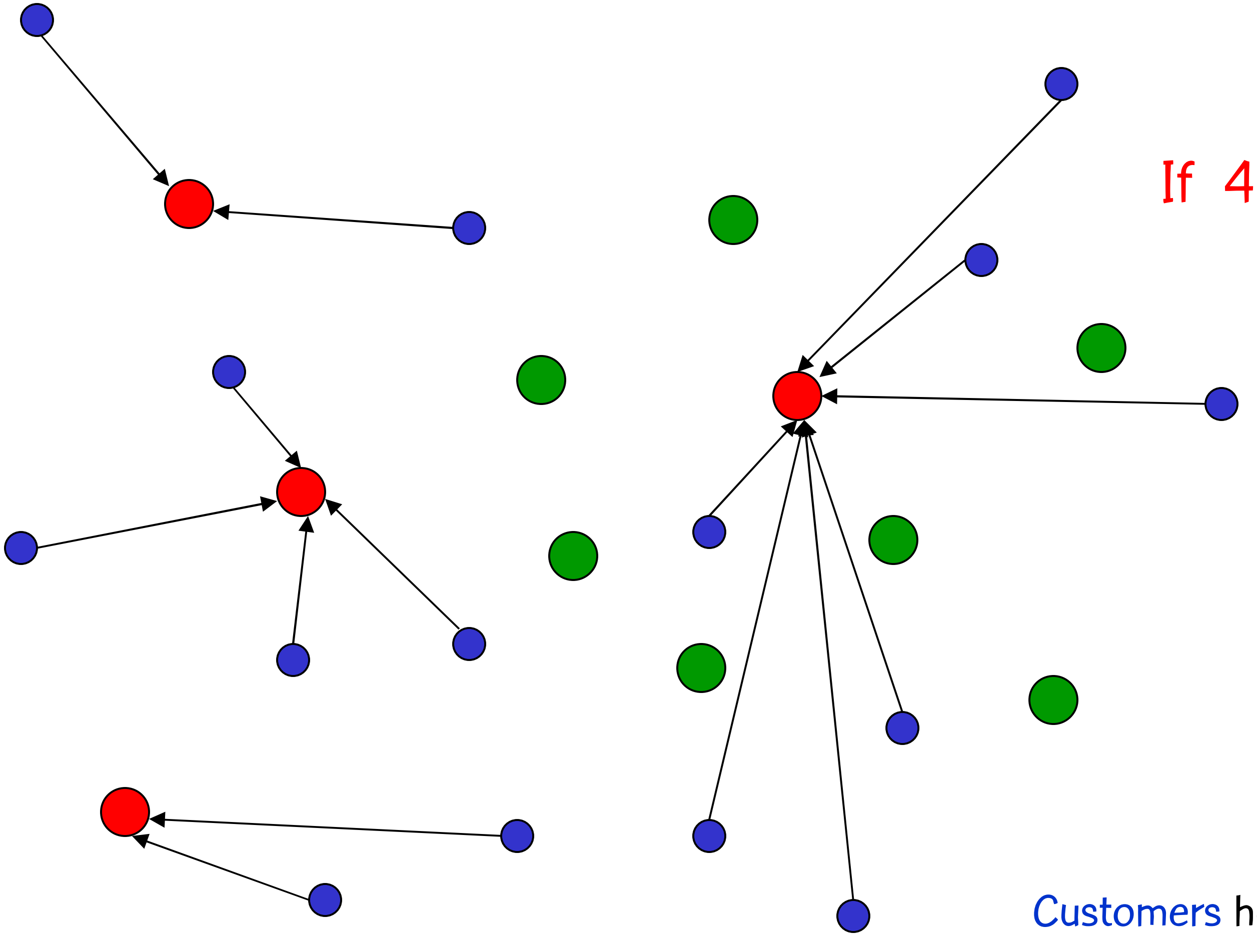


If 4 facilities are opened

n (=11) potential facility locations

m (=15) customers

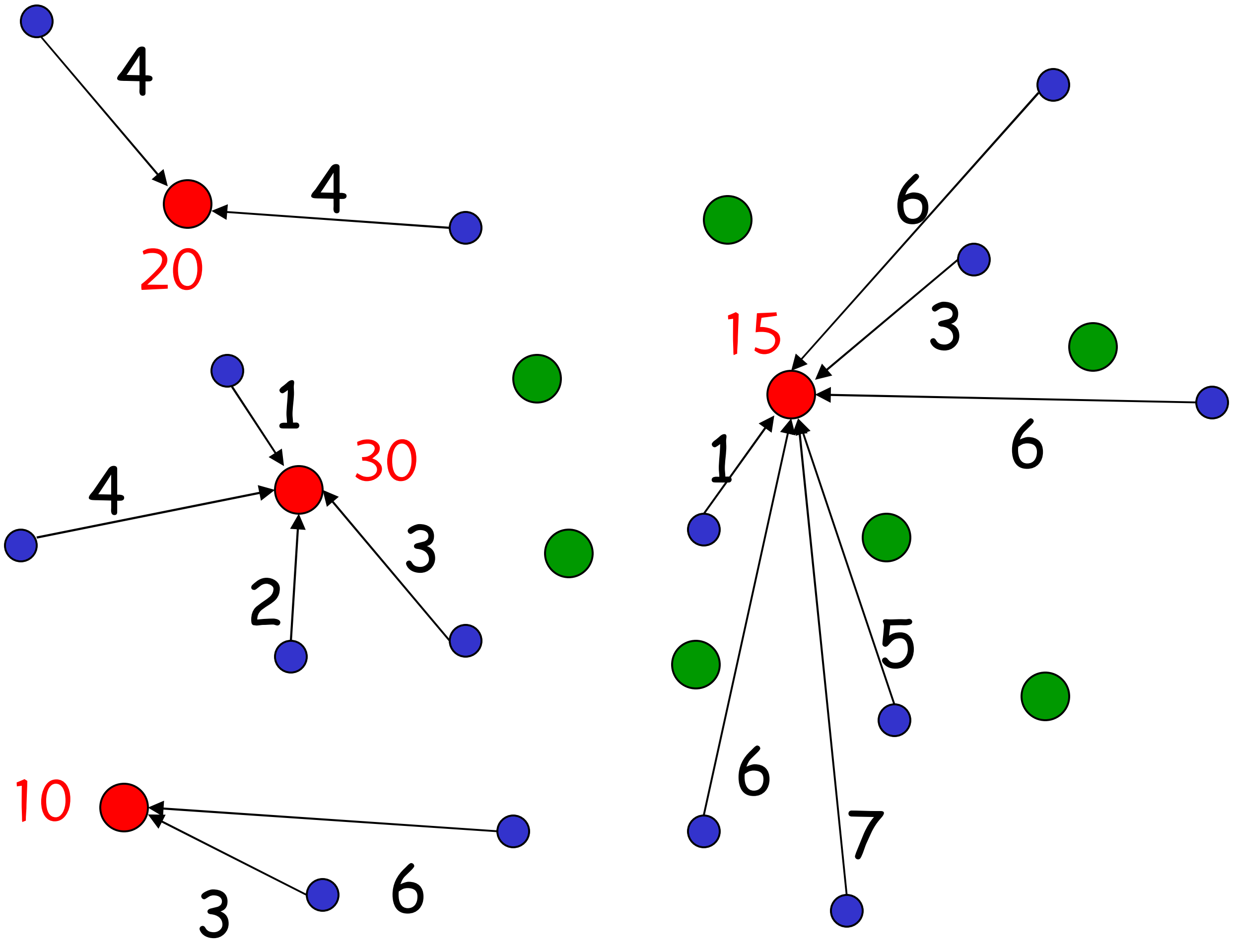
Uncapacitated facility location problem



If 4 facilities are opened

Customers home into nearest open facility.

Uncapacitated facility location problem

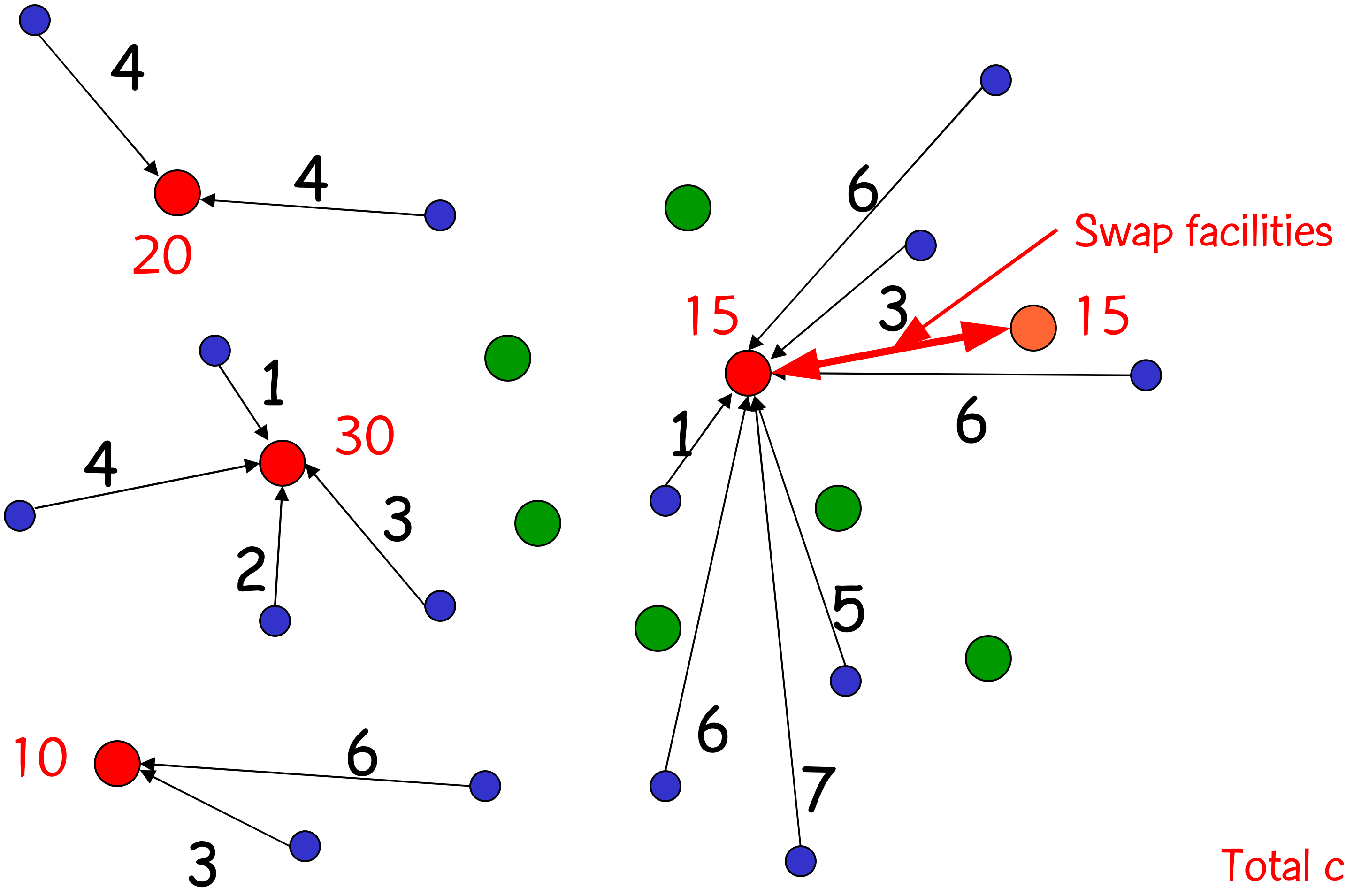


Objective of optimization:

Minimize sum of the distances between customers and their nearest open facility plus the cost of opening the facilities.

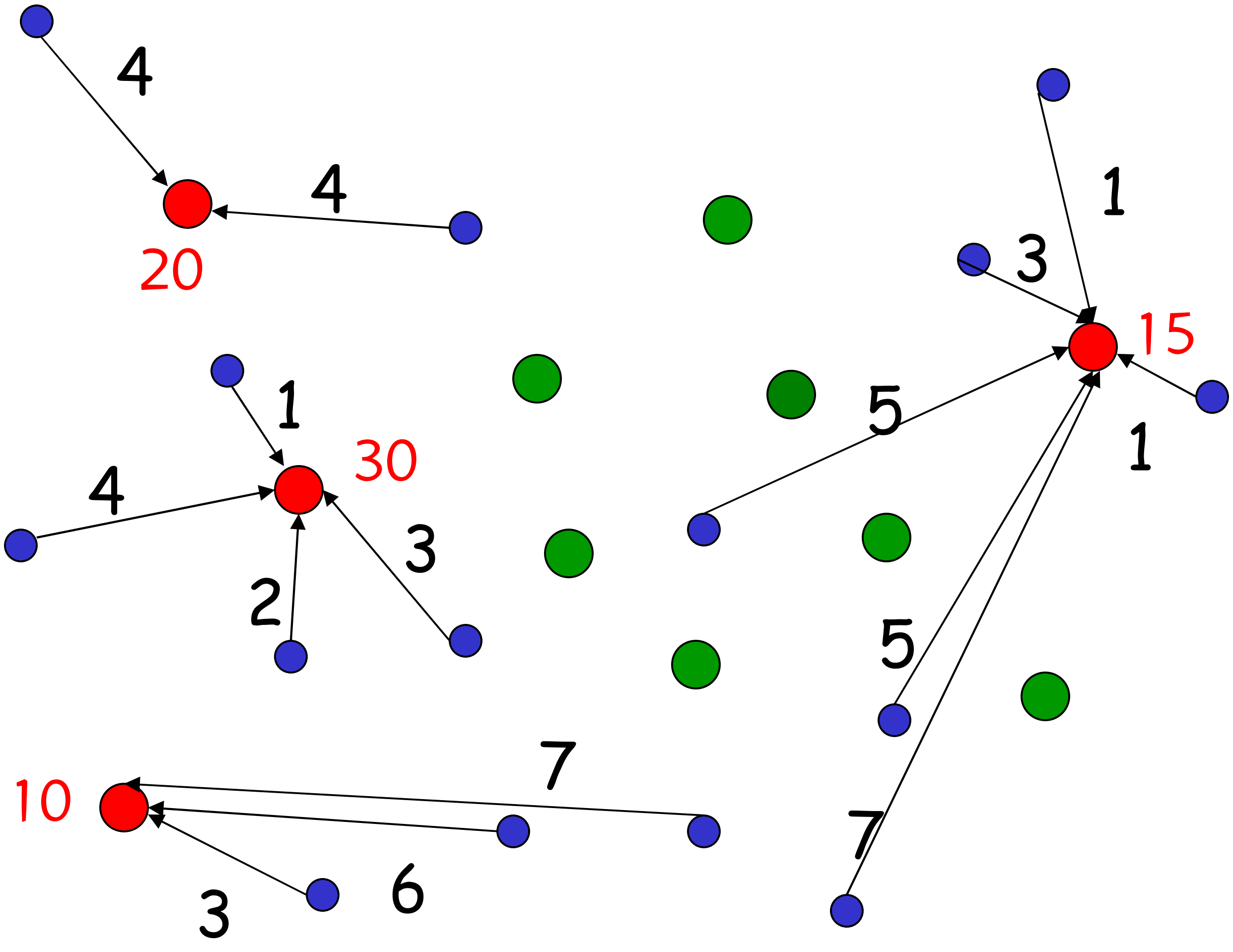
Total cost = 61+ 75 = 136

Uncapacitated facility location problem



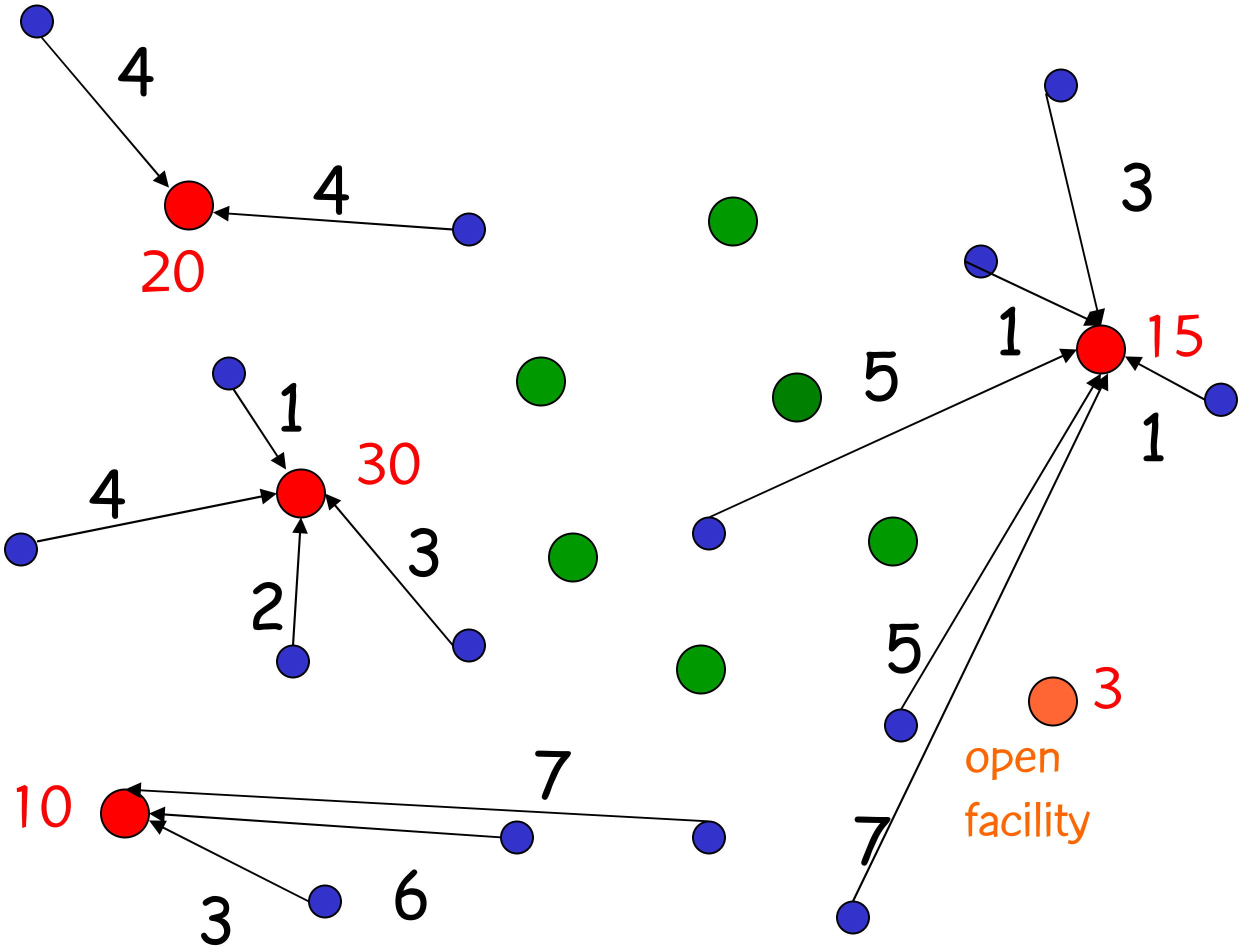
Total cost = 61+ 75 = 136

Uncapacitated facility location problem



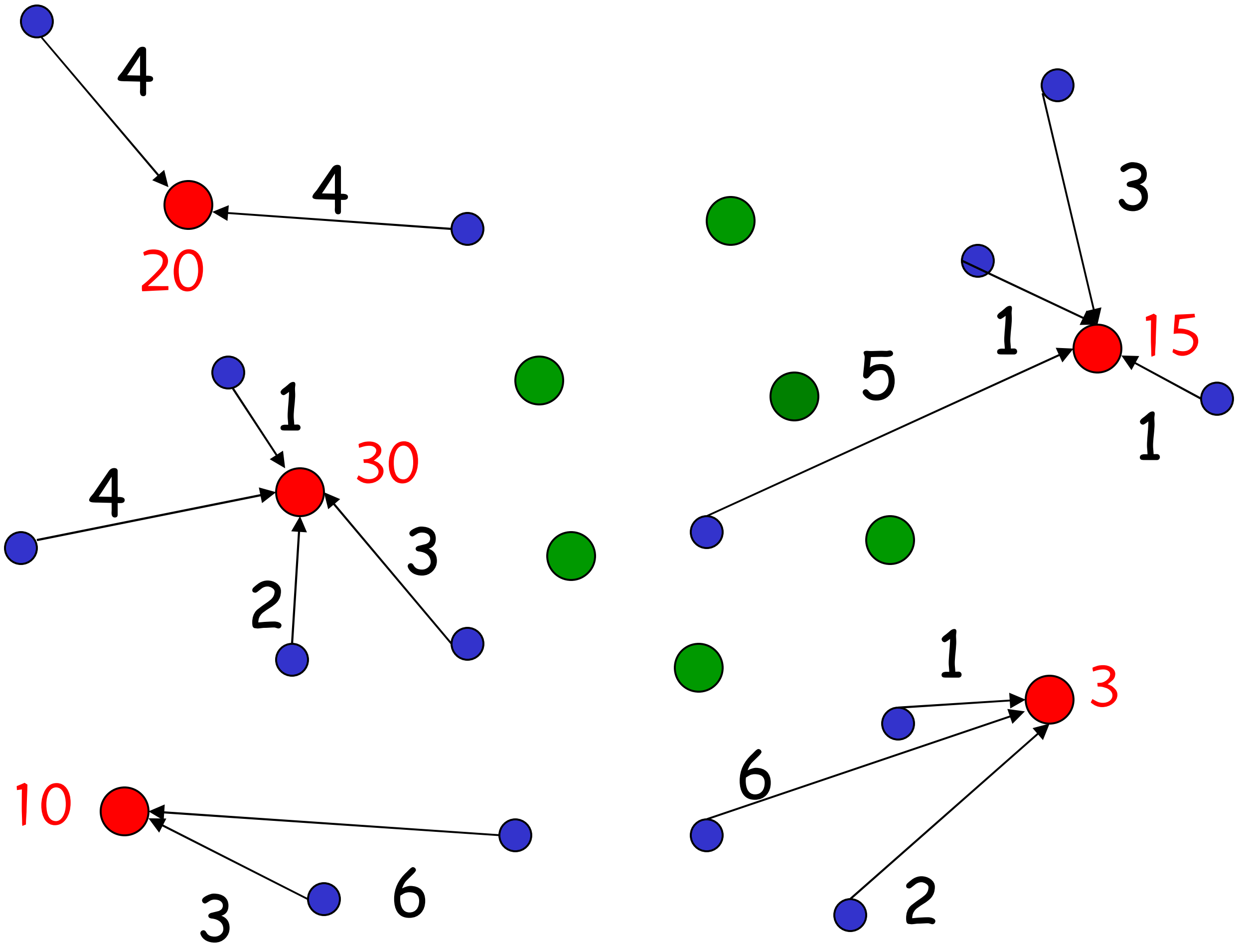
Total cost = 58 + 75 = 133 < 136

Uncapacitated facility location problem



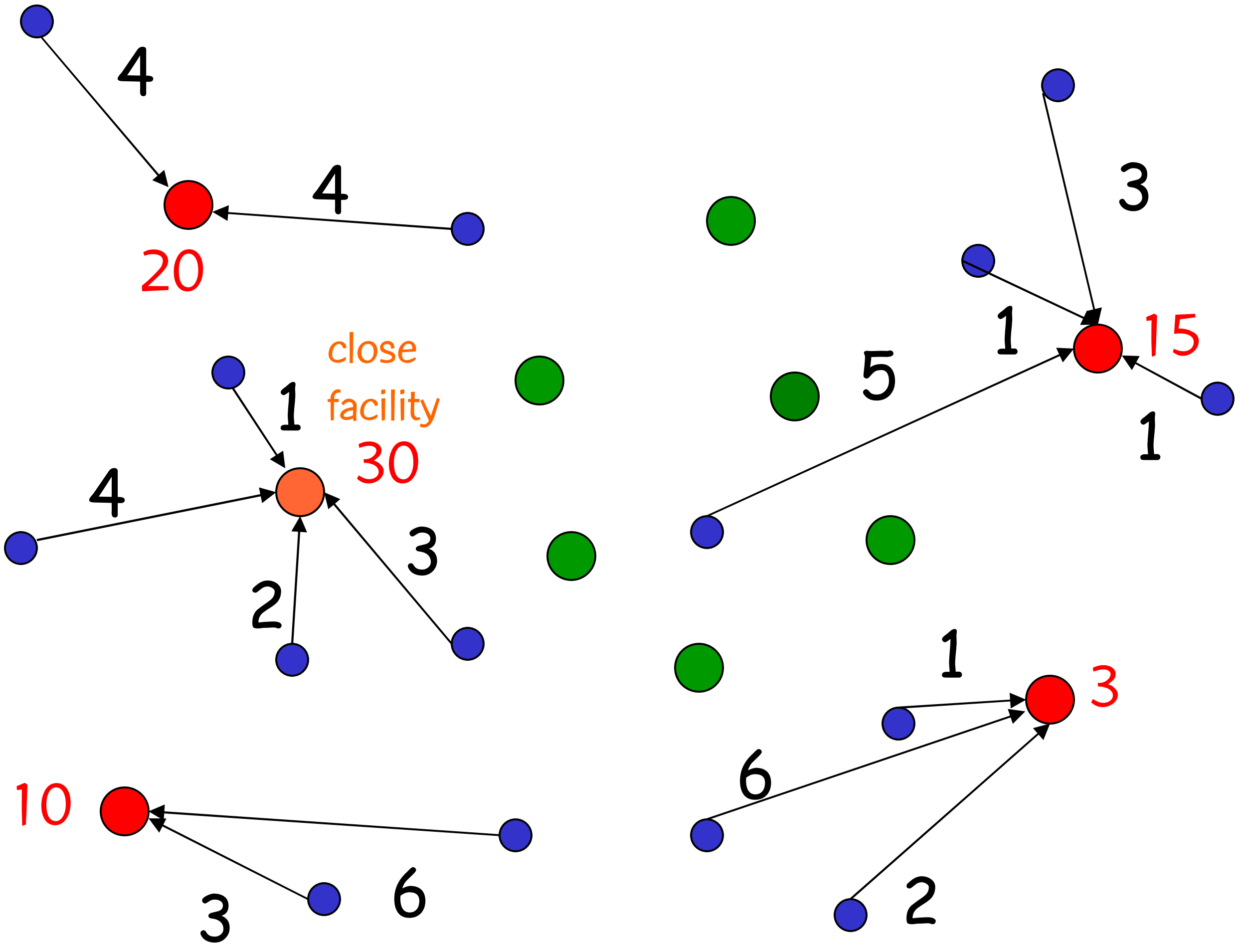
Total cost = 58 + 75 =
133 < 136

Uncapacitated facility location problem



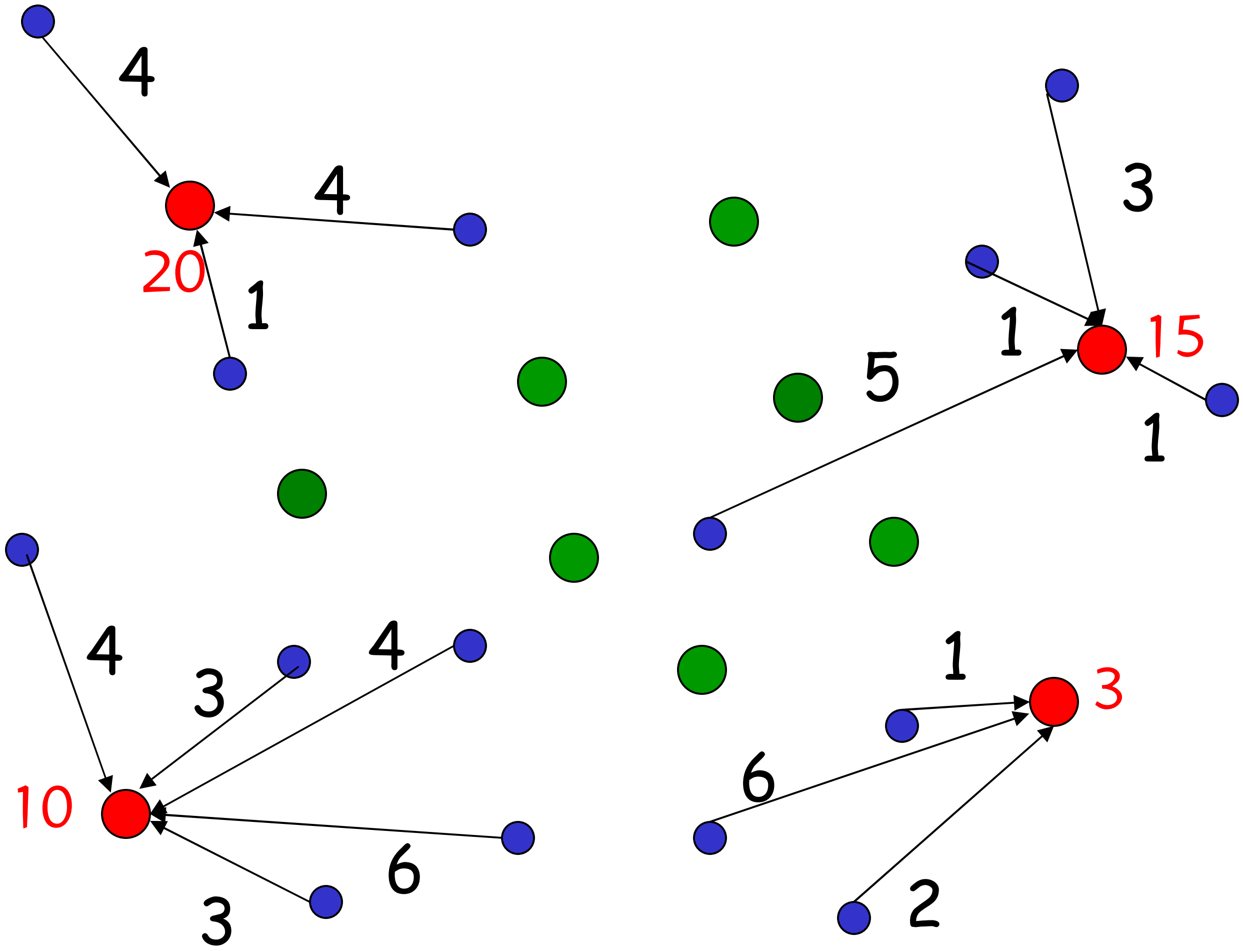
Total cost = 46 + 78 =
124 < 133

Uncapacitated facility location problem



Total cost = 46 + 78 = 124 < 133

Uncapacitated facility location problem



Total cost = 48 + 48 =
96 < 124

Set F of *potential facilities*, each with a setup cost $c(f)$.

Set U of users that must be served by a facility. The cost of servicing user u by facility f is $d(u, f)$.

Facility location problem: Determine a set of facilities $S \subseteq F$ to open so as to minimize the total cost:

$$\text{cost}(S) = \sum_{f \in S} c(f) + \sum_{u \in U} \min_{f \in S} d(u, f).$$

Uncapacitated facility location

- Customers home in to nearest open facility
- No limit on number of open facilities
- NP hard [Cournéjols, Nemhauser, & Wolsey, 1990]
- Perhaps the most common location problem, studied widely in literature both in theory & practice

Uncapacitated facility location

- Customers home in to nearest open facility
- No limit on number of open facilities
- NP hard [Cournéjols, Nemhauser, & Wolsey, 1990]
- Perhaps the most common location problem, studied widely in literature both in theory & practice

Uncapacitated facility location

- Customers home in to nearest open facility
- No limit on number of open facilities
- NP hard [Cournéjols, Nemhauser, & Wolsey, 1990]
- Perhaps the most common location problem, studied widely in literature both in theory & practice

Uncapacitated facility location

- Customers home in to nearest open facility
- No limit on number of open facilities
- NP hard [Cournéjols, Nemhauser, & Wolsey, 1990]
- Perhaps the most common location problem, studied widely in literature both in theory & practice

Uncapacitated facility location

- Exact methods exist, e.g. [Conn and Cournéjols, 1990; Körkel, 1989]
- NP-hard nature makes heuristics a natural choice for larger instances
- Shmoys, Tardos, & Aardal (1997) present a 3.16-opt approximation algorithm
- Improvements, e.g. [Jain et al., 2002, 2003; Mahdian, Ye, & Zhang, 2002] have led to polynomial-time algorithms that find a solution within a factor of around 1.5 from the optimal.

Uncapacitated facility location

- Exact methods exist, e.g. [Conn and Cournéjols, 1990; Körkel, 1989]
- NP-hard nature makes heuristics a natural choice for larger instances
- Shmoys, Tardos, & Aardal (1997) present a 3.16-opt approximation algorithm
- Improvements, e.g. [Jain et al., 2002, 2003; Mahdian, Ye, & Zhang, 2002] have led to polynomial-time algorithms that find a solution within a factor of around 1.5 from the optimal.

Uncapacitated facility location

- Exact methods exist, e.g. [Conn and Cournéjols, 1990; Körkel, 1989]
- NP-hard nature makes heuristics a natural choice for larger instances
- Shmoys, Tardos, & Aardal (1997) present a 3.16-opt approximation algorithm
- Improvements, e.g. [Jain et al., 2002, 2003; Mahdian, Ye, & Zhang, 2002] have led to polynomial-time algorithms that find a solution within a factor of around 1.5 from the optimal.

Uncapacitated facility location

- Exact methods exist, e.g. [Conn and Cournéjols, 1990; Körkel, 1989]
- NP-hard nature makes heuristics a natural choice for larger instances
- Shmoys, Tardos, & Aardal (1997) present a 3.16-opt approximation algorithm
- Improvements, e.g. [Jain et al., 2002, 2003; Mahdian, Ye, & Zhang, 2002] have led to polynomial-time algorithms that find a solution within a factor of around 1.5 from the optimal.

Uncapacitated facility location

- Unfortunately, there is not much more room for improvement: Guha & Khuller (1999) established a lower bound of 1.463 for the approximation factor.
- In practice, approximation algorithms tend to be much closer for non-pathological instances: The 1.61-opt algorithm of Jain et al. (2003) was always within 2% of optimal in their experiments.
- Though interesting in theory, approximation algorithms are often outperformed in practice by more straightforward heuristics with no particular performance guarantees.

Uncapacitated facility location

- Unfortunately, there is not much more room for improvement: Guha & Khuller (1999) established a lower bound of 1.463 for the approximation factor.
- In practice, approximation algorithms tend to be much closer for non-pathological instances: The 1.61-opt algorithm of Jain et al. (2003) was always within 2% of optimal in their experiments.
- Though interesting in theory, approximation algorithms are often outperformed in practice by more straightforward heuristics with no particular performance guarantees.

Uncapacitated facility location

- Unfortunately, there is not much more room for improvement: Guha & Khuller (1999) established a lower bound of 1.463 for the approximation factor.
- In practice, approximation algorithms tend to be much closer for non-pathological instances: The 1.61-opt algorithm of Jain et al. (2003) was always within 2% of optimal in their experiments.
- Though interesting in theory, approximation algorithms are often outperformed in practice by more straightforward heuristics with no particular performance guarantees.

Uncapacitated facility location

- Pioneering work on heuristics: Kuehn & Hamburger (1963)
- Since then, more sophisticated heuristics have been applied:
  - Simulated annealing [Alves & Almeida, 1992]
  - Genetic algorithms [Kratica et al., 2001]
  - Tabu search [Ghosh, 2003; Michel & Van Hentenryck, 2003]
  - Complete local search with memory [Ghosh, 2003]
- Dual-based methods have also shown promising results:
  - Dual ascent [Erlenkotter, 1978]
  - Lagrangean dual ascent [Guignard, 1988]
  - Volume algorithm [Barahona & Chudak, 1999]

Uncapacitated facility location

- Pioneering work on heuristics: Kuehn & Hamburger (1963)
- Since then, more sophisticated heuristics have been applied:
  - Simulated annealing [Alves & Almeida, 1992]
  - Genetic algorithms [Kratica et al., 2001]
  - Tabu search [Ghosh, 2003; Michel & Van Hentenryck, 2003]
  - Complete local search with memory [Ghosh, 2003]
- Dual-based methods have also shown promising results:
  - Dual ascent [Erlenkotter, 1978]
  - Lagrangean dual ascent [Guignard, 1988]
  - Volume algorithm [Barahona & Chudak, 1999]

Uncapacitated facility location

- Pioneering work on heuristics: Kuehn & Hamburger (1963)
- Since then, more sophisticated heuristics have been applied:
  - Simulated annealing [Alves & Almeida, 1992]
  - Genetic algorithms [Kratica et al., 2001]
  - Tabu search [Ghosh, 2003; Michel & Van Hentenryck, 2003]
  - Complete local search with memory [Ghosh, 2003]
- Dual-based methods have also shown promising results:
  - Dual ascent [Erlenkotter, 1978]
  - Lagrangean dual ascent [Guignard, 1988]
  - Volume algorithm [Barahona & Chudak, 1999]

Uncapacitated facility location

- Hofer (2002) presented computational comparison of five methods:
  - JMS, an approximation algorithm of Jain et al. (2002)
  - MYZ, an approximation algorithm of Mahdian et al. (2002)
  - A swap-based local search
  - Tabu search of Michel & Van Hentenryck (2003)
  - Volume algorithm of Barahona & Chudack (1999)
- Hofer's conclusion: tabu search finds best solutions in reasonable time and is recommended to practitioners.

Uncapacitated facility location

- Hofer (2002) presented computational comparison of five methods:
  - JMS, an approximation algorithm of Jain et al. (2002)
  - MYZ, an approximation algorithm of Mahdian et al. (2002)
  - A swap-based local search
  - Tabu search of Michel & Van Hentenryck (2003)
  - Volume algorithm of Barahona & Chudack (1999)
- Hofer's conclusion: tabu search finds best solutions in reasonable time and is recommended to practitioners.

Our algorithm

- We provide an alternative that can be even better in practice.
- It is a hybrid multistart heuristic akin to the one we developed in Resende & Werneck (2004) for the p-median problem
- A series of minor adaptations is enough to build a very robust algorithm, capable of obtaining near-optimal solutions for a wide variety of instances of the facility location problem.

Our algorithm

- We provide an alternative that can be even better in practice.
- It is a hybrid multistart heuristic akin to the one we developed in Resende & Werneck (2004) for the p-median problem
- A series of minor adaptations is enough to build a very robust algorithm, capable of obtaining near-optimal solutions for a wide variety of instances of the facility location problem.

Our algorithm

- We provide an alternative that can be even better in practice.
- It is a hybrid multistart heuristic akin to the one we developed in Resende & Werneck (2004) for the p-median problem
- A series of minor adaptations is enough to build a very robust algorithm, capable of obtaining near-optimal solutions for a wide variety of instances of the facility location problem.

Our algorithm

- Works in two phases:
  - Multistart routine with intensification: Each iteration builds a randomized solution and applies local search to it. The resulting solution S is combined, in a process called path-relinking, with another solution from a set of elite solutions, resulting in S'. The algorithm tries to insert S and S' into the elite set.
  - Post-optimization: Solutions from the elite set are combined with each other in a process that hopefully results in better solutions.

Our algorithm

- Works in two phases:
  - Multistart routine with intensification: Each iteration builds a rondomized solution and applies local search to it. The resulting solution S is combined with a process called path-relinking with another solution from a set of elite solutions, resulting in S'. The algorithm tries to insert S and S' into the elite set.
  - Post-optimization: Solutions from the elite set are combined with each other in a process that hopefully results in better solutions.

Our algorithm

- Works in two phases:
  - Multistart routine with intensification: Each iteration builds a rondomized solution and applies local search to it. The resulting solution S is combined with a process called path-relinking with another solution from a set of elite solutions, resulting in S'. The algorithm tries to insert S and S' into the elite set.
  - Post-optimization: Solutions from the elite set are combined with each other in a process that hopefully results in better solutions.

HYBRID heuristic for location problems

```
function HYBRID ($seed, maxit, elitesize$)
    randomize($seed$);
    init($elite, elitesize$);
    for $i = 1$ to $maxit$ do
        $S \leftarrow$ randomizedBuild();
        $S \leftarrow$ localSearch($S$);
        $S' \leftarrow$ select($elite, S$);
        if ($S' \neq$ NULL) then
            $S' \leftarrow$ pathRelinking($S, S'$);
            add($elite, S'$);
        endif
        add($elite, S$);
    endfor
    $S \leftarrow$ postOptimize($elite$);
    return $S$;
end HYBRID
```

Reuse of p-median heuristic

- Although the HYBRID heuristic was originally proposed for the p-median problem, its framework can be applied to other problems: in this case, facility location.
- Recall that the p-median problem is very similar to facility location: the only difference is that instead of assigning costs to facilities, the p-median problem must specify p, the exact number of facilities to be opened.
- With minor adaptations, we can reuse several of the components used in Resende & Werneck (2004), such as the construction algorithms, local search, and path-relinking.

Reuse of p-median heuristic

- Although the HYBRID heuristic was originally proposed for the p-median problem, its framework can be applied to other problems: in this case, facility location.
- Recall that the p-median problem is very similar to facility location: the only difference is that instead of assigning costs to facilities, the p-median problem must specify p, the exact number of facilities to be opened.
- With minor adaptations, we can reuse several of the components used in Resende & Werneck (2004), such as the construction algorithms, local search, and path-relinking.

Reuse of p-median heuristic

- Although the HYBRID heuristic was originally proposed for the p-median problem, its framework can be applied to other problems: in this case, facility location.
- Recall that the p-median problem is very similar to facility location: the only difference is that instead of assigning costs to facilities, the p-median problem must specify p, the exact number of facilities to be opened.
- With minor adaptations, we can reuse several of the components used in Resende & Werneck (2004), such as the construction algorithms, local search, and path-relinking.

Construction heuristic

- At iteration i, we determine the number p_i of facilities to open.
  - For i = 1, $p_i = \lceil m/2 \rceil$;
  - For i > 1, we pick the average number of facilities opened in the first i – 1 iterations;
- We then execute procedure sample of the p-median variant of HYBRID:
  - At each step, choose $\lceil \log_2 (m/p_i) \rceil$ facilities uniformly at random and select the one that reduces the total cost the most.

Construction heuristic

- At iteration i, we determine the number p_i of facilities to open.
  - For i = 1, $p_i = \lceil m/2 \rceil$;
  - For i > 1, we pick the average number of facilities opened in the first i – 1 iterations;
- We then execute procedure sample of the p-median variant of HYBRID:
  - At each step, choose $\lceil \log_2 (m/p_i) \rceil$ facilities uniformly at random and select the one that reduces the total cost the most.

Local search

- Local search in p-median variant: given solution S, find two facilities $f_r \in S$, $f_i \notin S$ which, if swapped, leads to a better solution.
  - This keeps number of facilities constant.
  - We also allow pure insertions and pure deletions, as well as swaps.
- All possible insertions, deletions, and swaps are considered, and the best among those is performed.
- Local search stops (at local minimum) when no improving move exists.

Local search

- Local search in p-median variant: given solution S, find two facilities $f_r \in S$, $f_i \notin S$ which, if swapped, leads to a better solution.
  - This keeps number of facilities constant.
  - We also allow pure insertions and pure deletions, as well as swaps.
- All possible insertions, deletions, and swaps are considered, and the best among those is performed.
- Local search stops (at local minimum) when no improving move exists.

Local search

- Local search in p-median variant: given solution S, find two facilities $f_r \in S$, $f_i \notin S$ which, if swapped, leads to a better solution.
  - This keeps number of facilities constant.
  - We also allow pure insertions and pure deletions, as well as swaps.
- All possible insertions, deletions, and swaps are considered, and the best among those is performed.
- Local search stops (at local minimum) when no improving move exists.

Parameters

- Hybrid is a GRASP with EvPR at post-processing phase.
- Besides random number seed, HYBRID takes only two input parameters:
  - N: number of iterations
  - E: size of pool of elite solutions
- In standard version, we use N = 32 and E = 10.

Parameters

- Hybrid is a GRASP with EvPR at post-processing phase.
- Besides random number seed, HYBRID takes only two input parameters:
  - N: number of iterations
  - E: size of pool of elite solutions
- In standard version, we use N = 32 and E = 10.

Parameters

- Recall running time of multistart phase depends linearly on number of iterations N, whereas post-optimization depends (roughly) quadratically on the pool size E.
- There, if we want to multiply the average running time of the algorithm by some factor X, we just multiply N by X and E by sqrt(X), rounding off appropriately.

Parameters

- Recall running time of multistart phase depends linearly on number of iterations N, whereas post-optimization depends (roughly) quadratically on the pool size E.
- There, if we want to multiply the average running time of the algorithm by some factor X, we just multiply N by X and E by sqrt(X), rounding off appropriately.

Empirical results

Experimental setup

- Algorithm implemented in C++ and compiled with the SGI MIPSPro C++ compiler (v. 7.30) with flags –O3 –OPT:Olimit=6586
- Runs were done on an SGI Challenge with 28 196-MHz MIPS 10000 processors, but each execution was limited to a single processor
- All CPU times reported are measured by the getrusage function with a precision of 1/60 second
- Random number generator: Mersenne Twister (Matsumoto and Nishimura, 1998)

Empirical results

Experimental setup

- Algorithm implemented in C++ and compiled with the SGI MIPSPro C++ compiler (v. 7.30) with flags –O3 –OPT:Olimit=6586
- Runs were done on an SGI Challenge with 28 196-MHz MIPS 10000 processors, but each execution was limited to a single processor
- All CPU times reported are measured by the getrusage function with a precision of 1/60 second
- Random number generator: Mersenne Twister (Matsumoto and Nishimura, 1998)

Empirical results

Experimental setup

- Algorithm implemented in C++ and compiled with the SGI MIPSPro C++ compiler (v. 7.30) with flags –O3 –OPT:Olimit=6586
- Runs were done on an SGI Challenge with 28 196-MHz MIPS 10000 processors, but each execution was limited to a single processor
- All CPU times reported are measured by the getrusage function with a precision of 1/60 second
- Random number generator: Mersenne Twister (Matsumoto and Nishimura, 1998)

Empirical results

Experimental setup

- Algorithm implemented in C++ and compiled with the SGI MIPSPro C++ compiler (v. 7.30) with flags –O3 –OPT:Olimit=6586
- Runs were done on an SGI Challenge with 28 196-MHz MIPS R10000 processors, but each execution was limited to a single processor
- All CPU times reported are measured by the getrusage function with a precision of 1/60 second
- Random number generator: Mersenne Twister (Matsumoto and Nishimura, 1998)

Empirical results

Test problems

- Algorithm was tested on all classes from UflLib (Hoefer, 2003) and on class GHOSH, described in Ghosh (2003).
- In every case, the number of users and potential facilities is the same (locations are the same).

http://www.mpi-sb.mpg.de/units/ag1/projects/benchmarks/UflLib

Empirical results

Test problems

- Algorithm was tested on all classes from UflLib (Hoefer, 2003) and on class GHOSH, described in Ghosh (2003).
- In every case, the number of users and potential facilities is the same (locations are the same).

http://www.mpi-sb.mpg.de/units/ag1/projects/benchmarks/UflLib

Instance class	Reference	Instances/Size	Notes
BK	Bilde & Krarup (1977)	200 instances, 30 to 100 users	d ~ [0,1000] $c \geq 1000$
FPP	Kochetov (2003)	80 instances, 133 & 307 users	Meant to be challenging for algorithms based on local search.
GAP	Kochetov (2003)	120 instances, 100 users	Large duality gaps. Hard for dual-based method.
GHOSH	Ghosh (2003)	90 instances, 250, 500, & 750 users	d ~ [1000,2000] A: c ~ [100,200] B: c ~ [1000,2000] C: c ~ [10000,20000]

Test problems

BK used in Hoefer's comparative analysis.

Instance class	Reference	Instances/Size	Notes
GR	Galvão & Raggi (1989)	50 instances, 50 to 200 users	d ~ shortest paths given as matrices
M*	Kratica et al. (2001)	22 instances, 100 to 2000 users	Meant to be close to real-life applications: many near-optimal solutions.
MED	Ahn et al. (1998); Barahona & Chudak (1999)	18 instances, 500 to 3000 users	Random points in unit square, Euclidean distances with 4 signif. digits.
ORLIB	Beasley (1993)	15 instances, 50 to 1000 users	Instances originally proposed for capacitated facility location problems.

Test problems

GR, M*, MED, and ORLIB used in Hoefer's comparative analysis.

Empirical results

Quality assessment

- Standard version of algorithm
- Run ten times on each instance with ten random number seeds (1,...,10)
- Compare to optima for FPP, GAP, BK, GR, and ORLIB and best upper bounds for MED and M*
- Geometric means given for times.

Class	Avg % dev	Time (secs)
BK	0.001	0.28
FPP	27.999	7.36
GAP	5.935	1.63
GHOSH	(0.039)	30.66
GR	0.000	0.31
M*	0.000	7.45
MED	(0.392)	284.88
ORLIB	0.000	0.18

Empirical results

Quality assessment

- On all five classes in Hoefer's analysis, our algorithms do very well.
- Matches best known bounds (usually optima) on GR, M*, and ORLIB.
- Few unlucky runs on class BK.
- On MED, solutions were on average 0.4% better than best known bounds
- Did well on GHOSH, compared to two algorithms.

Class	Avg % dev	Time (secs)
BK	0.001	0.28
FPP	27.999	7.36
GAP	5.935	1.63
GHOSH	(0.039)	30.66
GR	0.000	0.31
M*	0.000	7.45
MED	(0.392)	284.88
ORLIB	0.000	0.18

Empirical results

Quality assessment

- On all five classes in Hoefer's analysis, our algorithms do very well.
- Matches best known bounds (usually optima) on GR, M*, and ORLIB.
- Few unlucky runs on class BK.
- On MED, solutions were on average 0.4% better than best known bounds
- Did well on GHOSH, compared to two algorithms.

Class	Avg % dev	Time (secs)
BK	0.001	0.28
FPP	27.999	7.36
GAP	5.935	1.63
GHOSH	(0.039)	30.66
GR	0.000	0.31
M*	0.000	7.45
MED	(0.392)	284.88
ORLIB	0.000	0.18

Empirical results

Quality assessment

- On all five classes in Hoefer's analysis, our algorithms do very well.
- Matches best known bounds (usually optima) on GR, M*, and ORLIB.
- Few unlucky runs on class BK.
- On MED, solutions were on average 0.4% better than best known bounds
- Did well on GHOSH, compared to two algorithms.

Class	Avg % dev	Time (secs)
BK	0.001	0.28
FPP	27.999	7.36
GAP	5.935	1.63
GHOSH	(0.039)	30.66
GR	0.000	0.31
M*	0.000	7.45
MED	(0.392)	284.88
ORLIB	0.000	0.18

Empirical results

Quality assessment

- On all five classes in Hoefer's analysis, our algorithms do very well.
- Matches best known bounds (usually optima) on GR, M*, and ORLIB.
- Few unlucky runs on class BK.
- On MED, solutions were on average 0.4% better than best known bounds
- Did well on GHOSH, compared to two algorithms.

Class	Avg % dev	Time (secs)
BK	0.001	0.28
FPP	27.999	7.36
GAP	5.935	1.63
GHOSH	(0.039)	30.66
GR	0.000	0.31
M*	0.000	7.45
MED	(0.392)	284.88
ORLIB	0.000	0.18

Empirical results

Quality assessment

- On all five classes in Hoefer's analysis, our algorithms do very well.
- Matches best known bounds (usually optima) on GR, M*, and ORLIB.
- Few unlucky runs on class BK.
- On MED, solutions were on average 0.4% better than best known bounds
- Did well on GHOSH, compared to two algorithms.

Class	Avg % dev	Time (secs)
BK	0.001	0.28
FPP	27.999	7.36
GAP	5.935	1.63
GHOSH	(0.039)	30.66
GR	0.000	0.31
M*	0.000	7.45
MED	(0.392)	284.88
ORLIB	0.000	0.18

Empirical results

Quality assessment

- The remaining two classes: FPP & GAP were created with the intent of being hard.
- Solutions are much worse than for other classes.
- However, we show later that, if given more time, our algorithm can do well on these classes, too.

Class	Avg % dev	Time (secs)
BK	0.001	0.28
FPP	27.999	7.36
GAP	5.935	1.63
GHOSH	(0.039)	30.66
GR	0.000	0.31
M*	0.000	7.45
MED	(0.392)	284.88
ORLIB	0.000	0.18

Empirical results

Quality assessment

- The remaining two classes: FPP & GAP were created with the intent of being hard.
- Solutions are much worse than for other classes.
- However, we show later that, if given more time, our algorithm can do well on these classes, too.

Class	Avg % dev	Time (secs)
BK	0.001	0.28
FPP	27.999	7.36
GAP	5.935	1.63
GHOSH	(0.039)	30.66
GR	0.000	0.31
M*	0.000	7.45
MED	(0.392)	284.88
ORLIB	0.000	0.18

Empirical results

Quality assessment

- The remaining two classes: FPP & GAP were created with the intent of being hard.
- Solutions are much worse than for other classes.
- However, we show later that, if given more time, our algorithm can do well on these classes, too.

Class	Avg % dev	Time (secs)
BK	0.001	0.28
FPP	27.999	7.36
GAP	5.935	1.63
GHOSH	(0.039)	30.66
GR	0.000	0.31
M*	0.000	7.45
MED	(0.392)	284.88
ORLIB	0.000	0.18

Empirical results

Comparative analysis

- We have seen that our algorithm produces very good quality solutions on most of the classes of instances tested.
- On there own, however, these results don't mean much.
- Any reasonably scalable algorithm, given enough time, should be able to find good solutions.
- With this in mind: we compare our algorithm with the best algorithm from Hoefer's analysis: the tabu search of Michel and Van Hentenryck (2003)

Empirical results

Comparative analysis

- We have seen that our algorithm produces very good quality solutions on most of the classes of instances tested.
- On their own, however, these results don't mean much.
- Any reasonably scalable algorithm, given enough time, should be able to find good solutions.
- With this in mind: we compare our algorithm with the best algorithm from Hoefer's analysis: the tabu search of Michel and Van Hentenryck (2003)

Empirical results

Comparative analysis

- We have seen that our algorithm produces very good quality solutions on most of the classes of instances tested.
- On there own, however, these results don't mean much.
- Any reasonably scalable algorithm, given enough time, should be able to find good solutions.
- With this in mind: we compare our algorithm with the best algorithm from Hoefer's analysis: the tabu search of Michel and Van Hentenryck (2003)

Empirical results

Comparative analysis

- We have seen that our algorithm produces very good quality solutions on most of the classes of instances tested.
- On there own, however, these results don't mean much.
- Any reasonably scalable algorithm, given enough time, should be able to find good solutions.
- With this in mind: we compare our algorithm with the best algorithm from Hoefer's analysis: the tabu search of Michel and Van Hentenryck (2003)

- We downloaded TABU from UflLib and ran it on our computer with 500 iterations (as in Hoefer's experiments).
- Since TABU was faster than our standard version, we compare with a faster HYBRID with N = 8 and E = 5.
- Both algorithms were run 10 times on each instance

Class	HYBRID		TABU	
	%dev	time	%dev	time
BK	.028	0.082	0.076	0.152
FPP	66.49	1.730	97.06	0.604
GAP	9.502	0.369	16.50	0.244
GHOSH	(0.032)	7.887	0.002	4.621
GR	0.000	0.087	0.103	0.158
M*	0.004	2.087	0.011	1.615
MED	(0.369)	75.231	0.073	69.552
ORLIB	0.000	0.046	0.024	0.155

time in seconds (196 MHz R10000)

- We downloaded TABU from UflLib and ran it on our computer with 500 iterations (as in Hoefer's experiments).
- Since TABU was faster than our standard version, we compare with a faster HYBRID with N = 8 and E = 5.
- Both algorithms were run 10 times on each instance

Class	HYBRID		TABU	
	%dev	time	%dev	time
BK	.028	0.082	0.076	0.152
FPP	66.49	1.730	97.06	0.604
GAP	9.502	0.369	16.50	0.244
GHOSH	(0.032)	7.887	0.002	4.621
GR	0.000	0.087	0.103	0.158
M*	0.004	2.087	0.011	1.615
MED	(0.369)	75.231	0.073	69.552
ORLIB	0.000	0.046	0.024	0.155

time in seconds (196 MHz R10000)

- We downloaded TABU from UflLib and ran it on our computer with 500 iterations (as in Hoefer's experiments).
- Since TABU was faster than our standard version, we compare with a faster HYBRID with N = 8 and E = 5.
- Both algorithms were run 10 times on each instance

	HYBRID		TABU	
Class	%dev	time	%dev	time
BK	.028	0.082	0.076	0.152
FPP	66.49	1.730	97.06	0.604
GAP	9.502	0.369	16.50	0.244
GHOSH	(0.032)	7.887	0.002	4.621
GR	0.000	0.087	0.103	0.158
M*	0.004	2.087	0.011	1.615
MED	(0.369)	75.231	0.073	69.552
ORLIB	0.000	0.046	0.024	0.155

time in seconds (196 MHz R10000)

- Both algorithms had similar running times.
- Even though running times are much lower than for standard version of HYBRID, both algorithms find very good quality solutions on five classes in Hoefer's analysis.
- On classes FPP, GAP, & MED, however, HYBRID does better than TABU.
- Time spent on classes FPP and GAP is only about one second.

	HYBRID		TABU	
Class	%dev	time	%dev	time
BK	.028	0.082	0.076	0.152
FPP	66.49	1.730	97.06	0.604
GAP	9.502	0.369	16.50	0.244
GHOSH	(0.032)	7.887	0.002	4.621
GR	0.000	0.087	0.103	0.158
M*	0.004	2.087	0.011	1.615
MED	(0.369)	75.231	0.073	69.552
ORLIB	0.000	0.046	0.024	0.155

time in seconds (196 MHz R10000)

- Both algorithms had similar running times.
- Even though running times are much lower than for standard version of HYBRID, both algorithms find very good quality solutions on five classes in Hoefer's analysis.
- On classes FPP, GAP, & MED, however, HYBRID does better than TABU.
- Time spent on classes FPP and GAP is only about one second.

Class	HYBRID		TABU	
	%dev	time	%dev	time
BK	.028	0.082	0.076	0.152
FPP	66.49	1.730	97.06	0.604
GAP	9.502	0.369	16.50	0.244
GHOSH	(0.032)	7.887	0.002	4.621
GR	0.000	0.087	0.103	0.158
M*	0.004	2.087	0.011	1.615
MED	(0.369)	75.231	0.073	69.552
ORLIB	0.000	0.046	0.024	0.155

time in seconds (196 MHz R10000)

- Both algorithms had similar running times.
- Even though running times are much lower than for standard version of HYBRID, both algorithms find very good quality solutions on five classes in Hoefer's anaylsis.
- On classes FPP, GAP, & MED, however, HYBRID does better than TABU.
- Time spent on classes FPP and GAP is only about one second.

	HYBRID		TABU	
Class	%dev	time	%dev	time
BK	.028	0.082	0.076	0.152
FPP	66.49	1.730	97.06	0.604
GAP	9.502	0.369	16.50	0.244
GHOSH	(0.032)	7.887	0.002	4.621
GR	0.000	0.087	0.103	0.158
M*	0.004	2.087	0.011	1.615
MED	(0.369)	75.231	0.073	69.552
ORLIB	0.000	0.046	0.024	0.155

time in seconds (196 MHz R10000)

- Both algorithms had similar running times.
- Even though running times are much lower than for standard version of HYBRID, both algorithms find very good quality solutions on five classes in Hoefer's analysis.
- On classes FPP, GAP, & MED, however, HYBRID does better than TABU.
- Time spent on classes FPP and GAP is only about one second.

	HYBRID		TABU	
Class	%dev	time	%dev	time
BK	.028	0.082	0.076	0.152
FPP	66.49	1.730	97.06	0.604
GAP	9.502	0.369	16.50	0.244
GHOSH	(0.032)	7.887	0.002	4.621
GR	0.000	0.087	0.103	0.158
M*	0.004	2.087	0.011	1.615
MED	(0.369)	75.231	0.073	69.552
ORLIB	0.000	0.046	0.024	0.155

time in seconds (196 MHz R10000)

Longer runs

- Both HYBRID and TABU should benefit if given more time to solve instances in GAP and FPP.
- We ran TABU with 1000, 2000, 4000, ..., 64000 iterations and HYBRID with N:E pairs 4:3, 8:5, 16:7, 32:10 (standard HYBRID), 64:14, 128:20, 256:28, and 512:40.

Longer runs

- Both HYBRID and TABU should benefit if given more time to solve instances in GAP and FPP.
- We ran TABU with 1000, 2000, 4000, …, 64000 iterations and HYBRID with N:E pairs 4:3, 8:5, 16:7, 32:10 (standard HYBRID), 64:14, 128:20, 256:28, and 512:40.

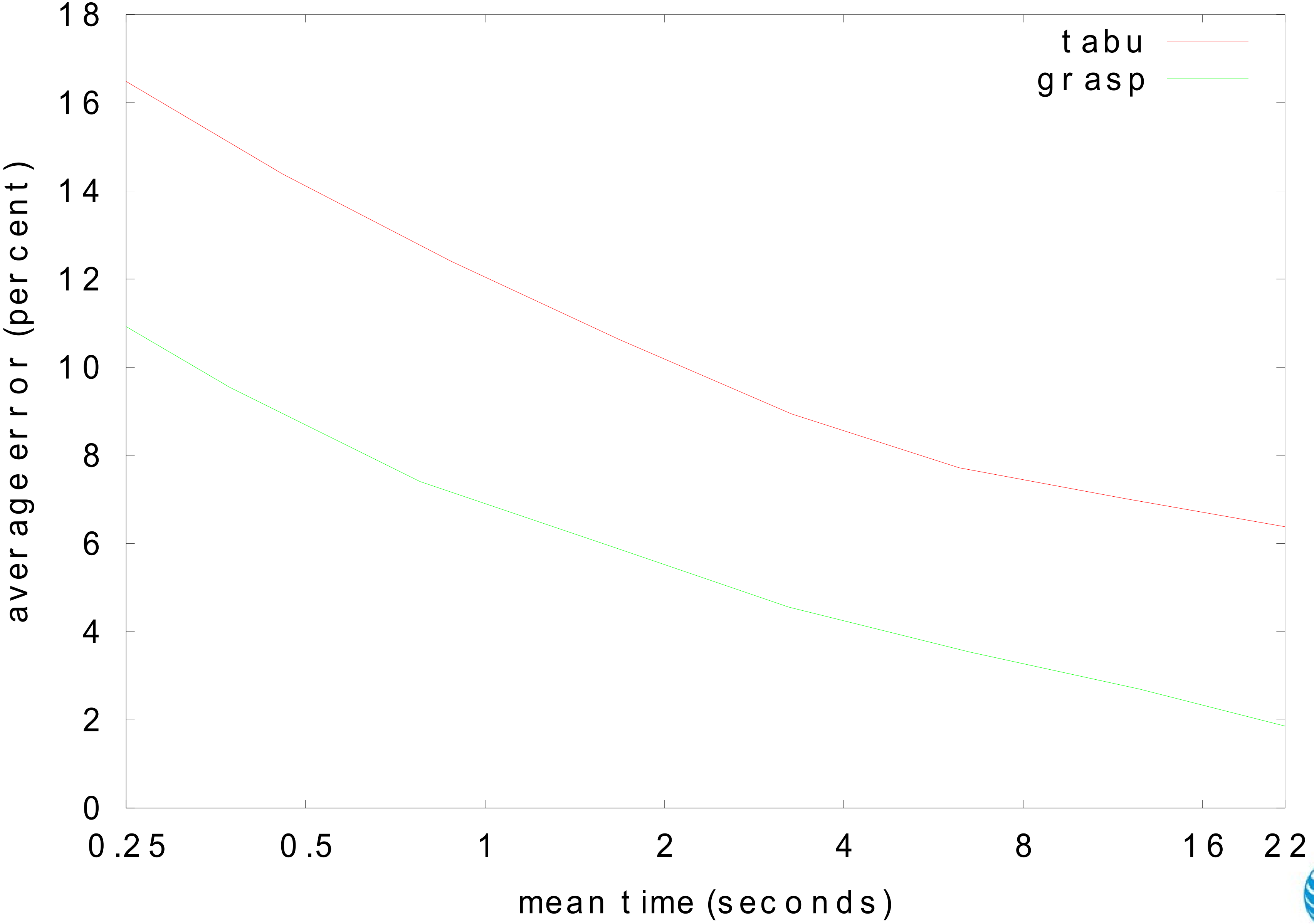
HYBRID				TABU		
iteration	elite	% error	time	iteration	% error	time
§4	3	12.961	0.14	§500	16.50	0.25
8	5	9.543	0.37	1000	14.38	0.46
16	7	7.407	0.78	2000	12.40	0.88
32	10	5.932	1.63	4000	10.62	0.88
64	14	4.561	3.23	8000	8.94	3.27
128	20	3.541	6.49	16000	7.72	6.24
256	28	2.700	12.54	32000	7.02	11.85
512	40	1.685	24.69	64000	6.35	22.62

Time in seconds (196MHz R10000)

Means over ten runs.

GAP class

GAP class



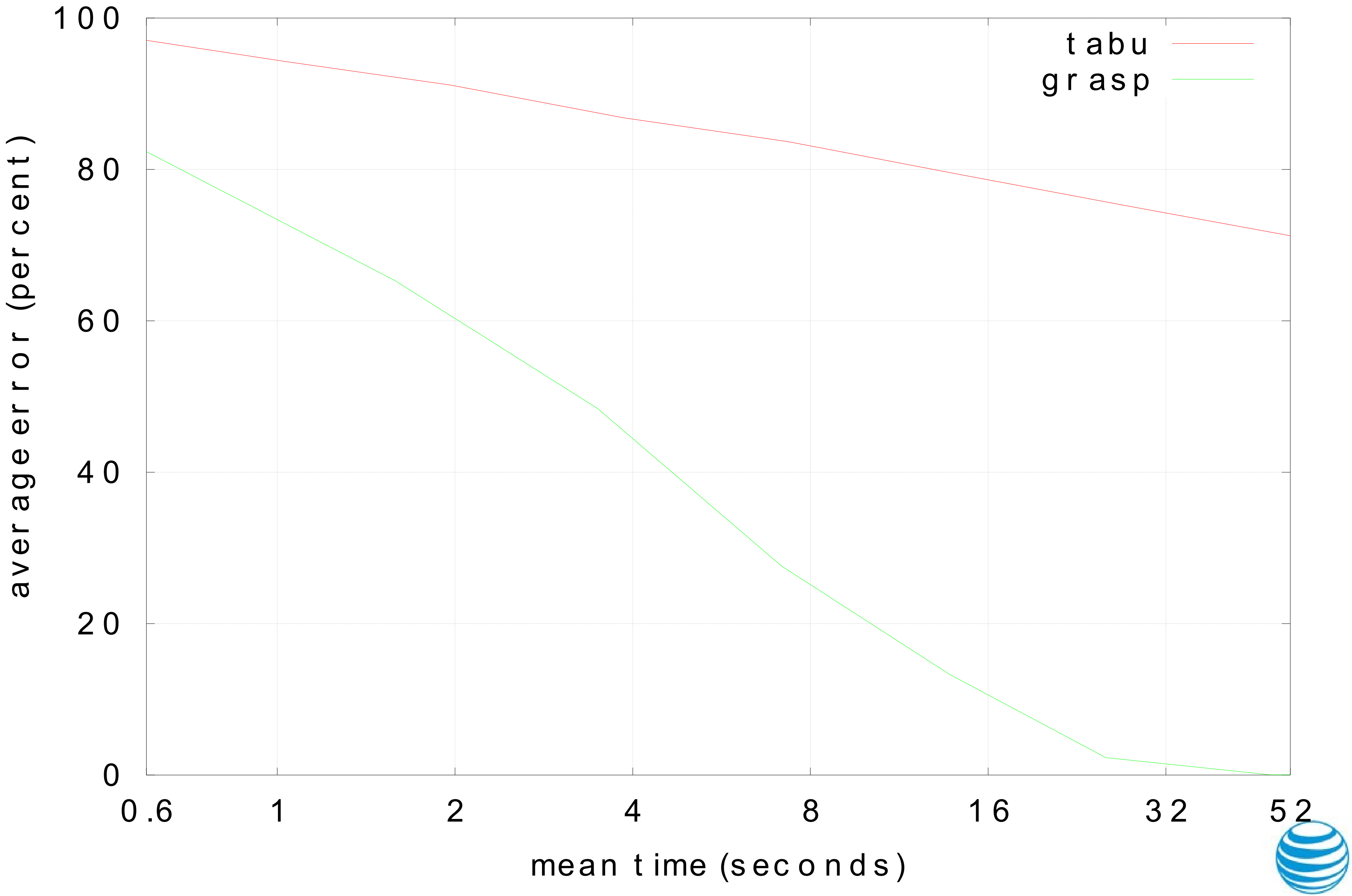
18
16
14
12
10
8
6
4
2
0
average error (percent)
tabu
grasp
0.25
0.5
1
2
4
8
16
22
mean time (seconds)

HYBRID				TABU		
iteration	elite	% error	time	iteration	% error	time
§4	3	82.832	0.58	§500	97.06	0.60
8	5	65.265	1.59	1000	94.22	1.04
16	7	48.413	3.49	2000	91.14	1.97
32	10	27.610	7.15	4000	86.81	3.86
64	14	13.279	13.79	8000	83.67	7.34
128	20	2.307	25.33	16000	79.32	14.34
256	28	0.018	48.17	32000	75.16	27.71
512	40	0.009	93.59	64000	71.15	52.60

Time in seconds (196MHz R10000)

Means over ten runs.

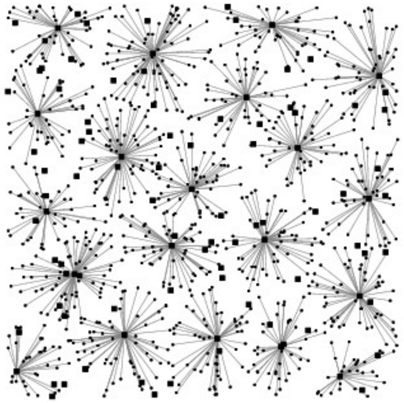
FPP class


100
80
60
40
20
0
average error (percent)
tabu
grasp
0.6
1
2
4
8
16
32
52
mean time (seconds)

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 (for research & academic use only) at:

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



Concluding remarks

- In this short course, we reviewed basic and advanced concept of GRASP for combinatorial optimization.
- We did not cover a recent development: C-GRASP, or Continuous GRASP, for solving general global optimization subject to box constraints.

Concluding remarks

http://mauricio.resende.info/MiniCursoGRASP.pdf

- The course book "An introduction to GRASP" covers all of the material presented in this course.
- Chapter 1 is an introduction to basic and advanced concepts of GRASP.
- Chapter 2 covers GRASP with path-relinking.
- Chapter 3 introduces GRASP with perturbations and hybridization with path-relinking and variable neighborhood search.

Concluding remarks

- Chapter 4 introduces GRASP with evolutionary path-relinking.
- Chapter 5 introduces TTT plots.
- Chapter 6 discusses the probability distribution of running time for GRASP.
- Chapter 7 considers parallel implementation of GRASP.

Concluding remarks

- Chapter 8 considers strategies for implementing GRASP with path-relinking.
- Chapter 9 presents parallel implementations of GRASP with path-relinking applied to job shop scheduling.
- Chapters 10, 11, and 12 show an example of an efficient implementation of GRASP for location problems.
- Chapter 13 is an updated annotated bibliography of GRASP.

The End

These slides and all papers cited in this short course can be downloaded from my homepage:
http://www.research.att.com/~mgcr

google.com search key: Mauricio Resende