CORK INSTITUTE OF TECHNOLOGY
INSTITIÚID TEICNEOLAÍOCHTA CHORCAÍ

Computing@CIT

Metaheuristic Optimization

Local Search

Dr. Diarmuid Grimes

Source for some of the local search material :
Pascal Van Henternyck's Coursera course on Discrete Optimisation
https://www.coursera.org/learn/discrete-optimization

Recap: Local Search, where are we?

When we reach an optima (i.e. a state for which no neighbor has an improving score) we

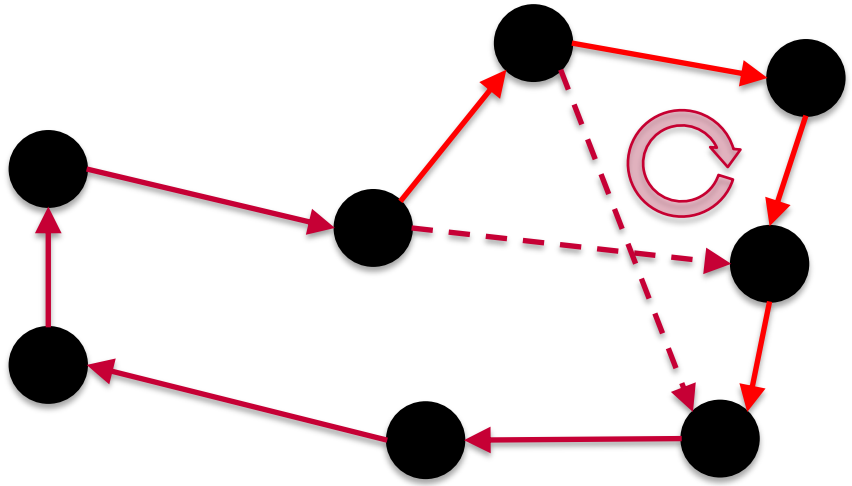
a) Don't know if this is a local or global optima

b) Don't even know if it is a (local) optima!

- Often we are checking only a subset of the neighborhood in each iteration
  - E.g. N-queens, choose a queen and assign it an improving value if one exists
  - If no improving value for that queen then we move to next iteration as there may be improving value for some other queen
  - But not storing information, no memory! So will forget that we have tried this queen already and may select it again in a future iteration but with same state
  - Similar can occur with sideways moves if a plateau

Recap: Local search for TSP: 2-OPT

- Task: find the shortest path to visit all cities exactly once
- Local Move:
  - Select two edges and replace them by two other edges



Recap: Local search for TSP: 2-OPT

- A TSP tour T is called 2-optimal if there is no 2-adjacent tour to T with lower cost than T.

- 2-opt heuristic:

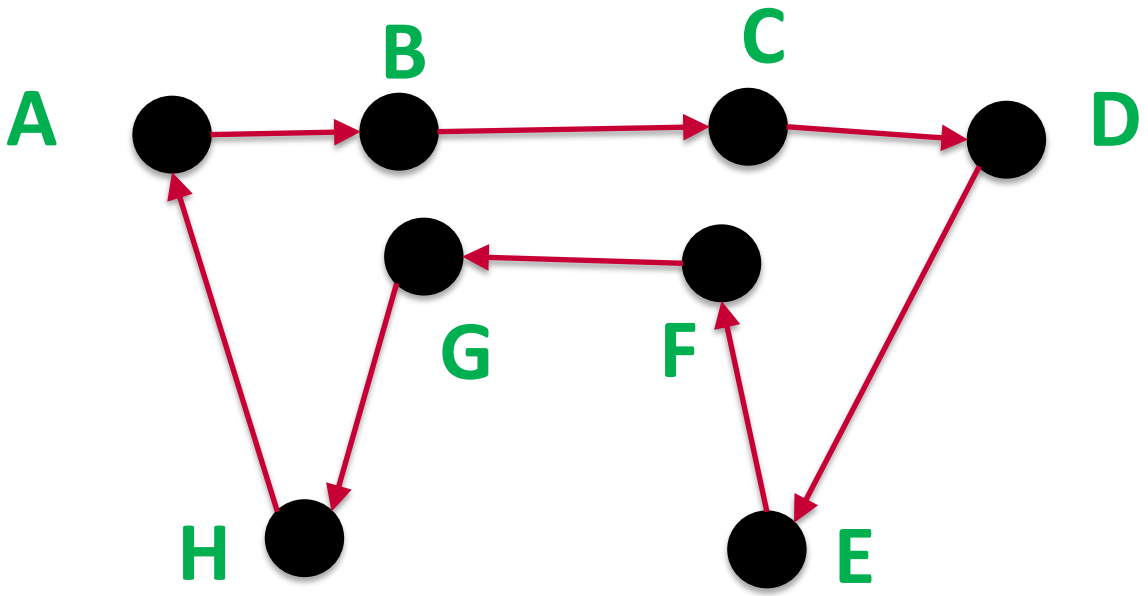
  Look for a 2-adjacent tour with lower cost than the current tour.

  If one is found then it replaces the current tour.

  This continues until there is a 2-optimal tour.

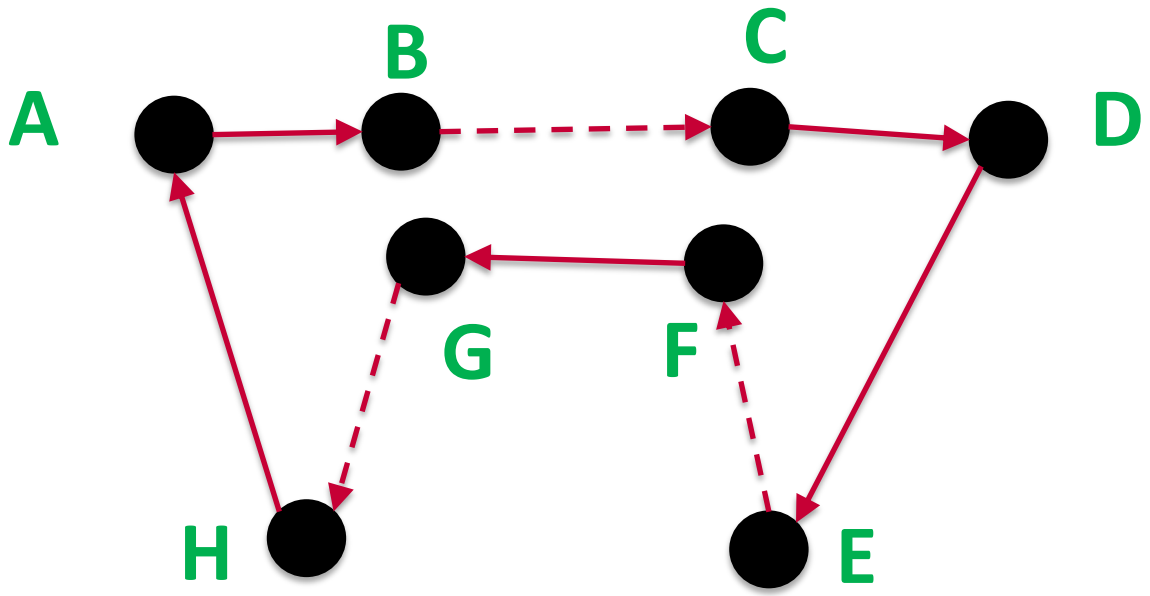
Recap: Local search for TSP: 3-OPT

- Task: find the shortest path to visit all cities exactly once
- Local Move:
  - Select three edges and replace them by three other edges
  - 7 different possibilities for replacement for valid tour (if including 3 2-opt moves!)



Recap: Local search for TSP: 3-OPT

- Task: find the shortest path to visit all cities exactly once
- Local Move:
  - Select three edges and replace them by three other edges
  - 7 different possibilities for replacement for valid tour (if including 3 2-opt moves!)



Recap: Local search for TSP: 3-OPT

ABCDEGFHA

ABEDCGFHA

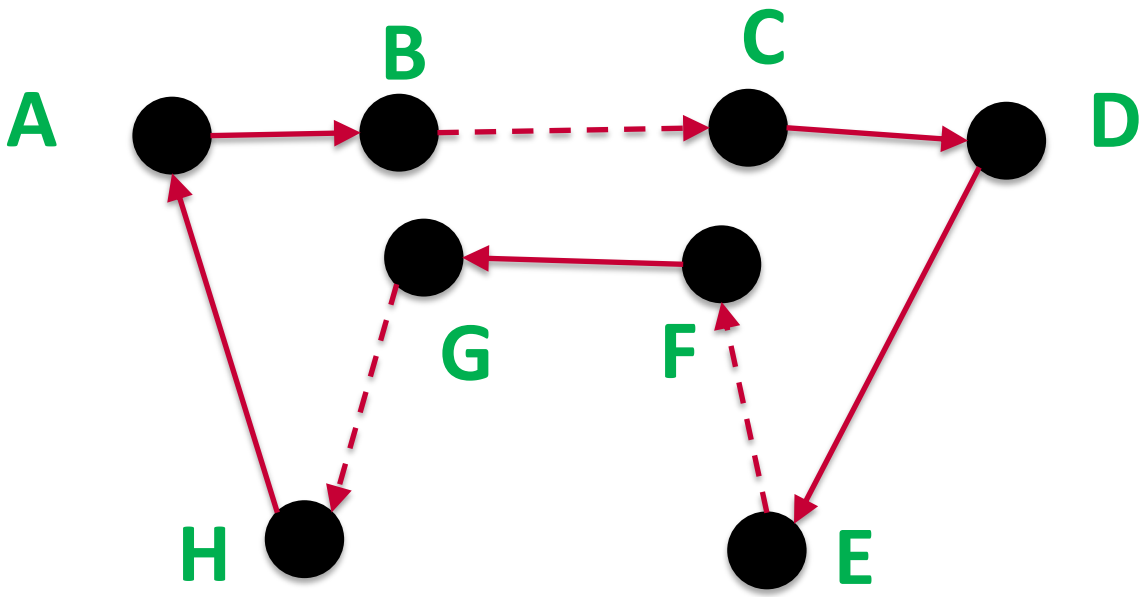
ABFGEDCHA

ABGFEDCHA

ABGFCDEHA

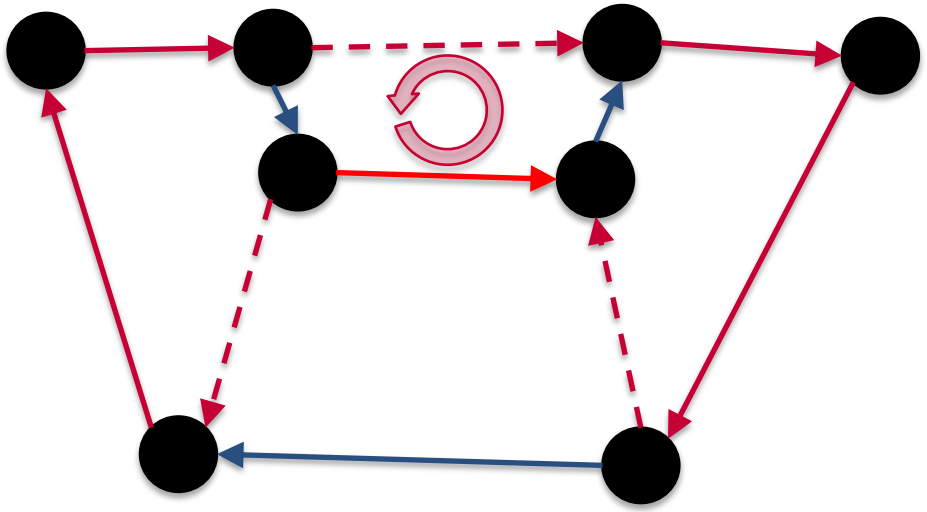
ABFGCDEHA

ABEDCFGHA



Recap: Local search for TSP: 3-OPT

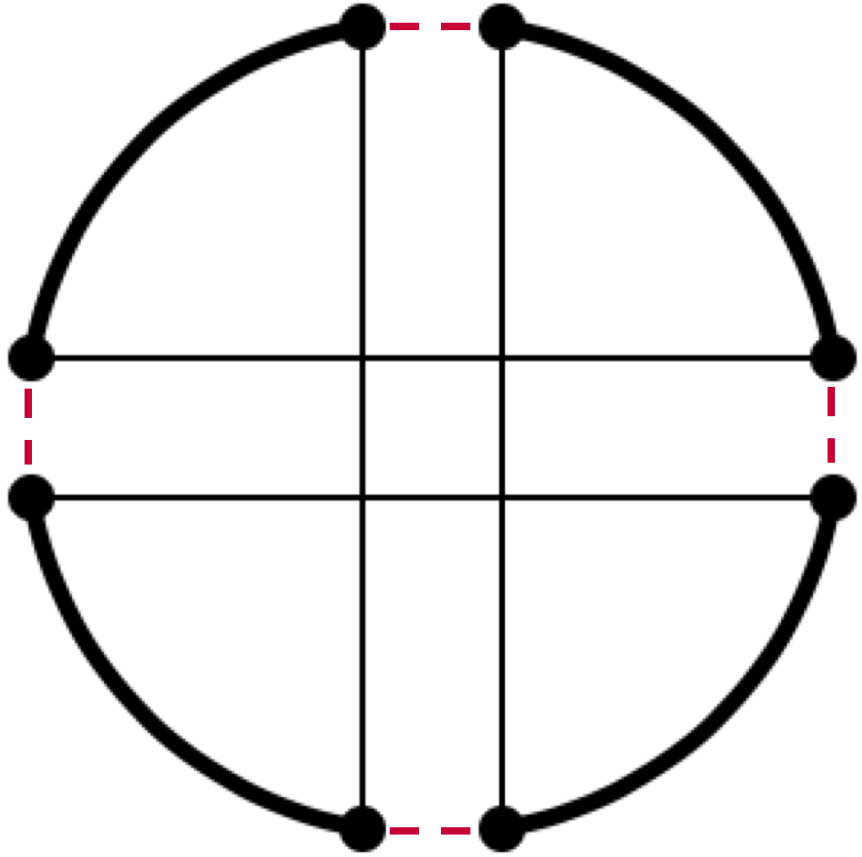
- Task: find the shortest path to visit all cities exactly once
- Local Move:
  - Select three edges and replace them by three other edges
  - 7 different possibilities for replacement for valid tour (including 3 2-opt moves!)



Recap: Local search for TSP

- 2-OPT:
  - Neighborhood is set of all tours reached by swapping two edges
- 3-OPT:
  - Neighborhood is set of all tours reached by swapping three edges
  - Much better than 2-OPT in quality but more expensive: $O(N^2)$ vs $O(N^3)$
- 4-OPT?
  - Neighborhood is set of all tours reached by swapping four edges
  - Marginally better but much more expensive $O(N^4)$
  - *Double bridge* move + 3-opt very popular

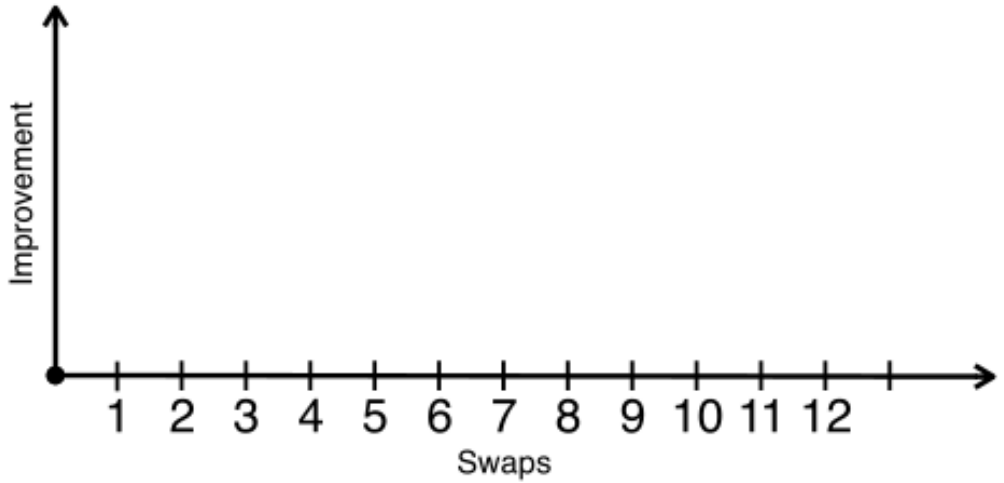
Recap: Local search for TSP: Double Bridge



Local search for TSP: K-Opt

- Task: find the shortest path to visit all cities exactly once
  - K-OPT (and Lin-Kernighan heuristic improved by Helsgaun for LKH-2):
    - Replace the notion of one favorable swap by a sequence of favorable swaps
    - Do not search for the entire set of sequences but build them incrementally

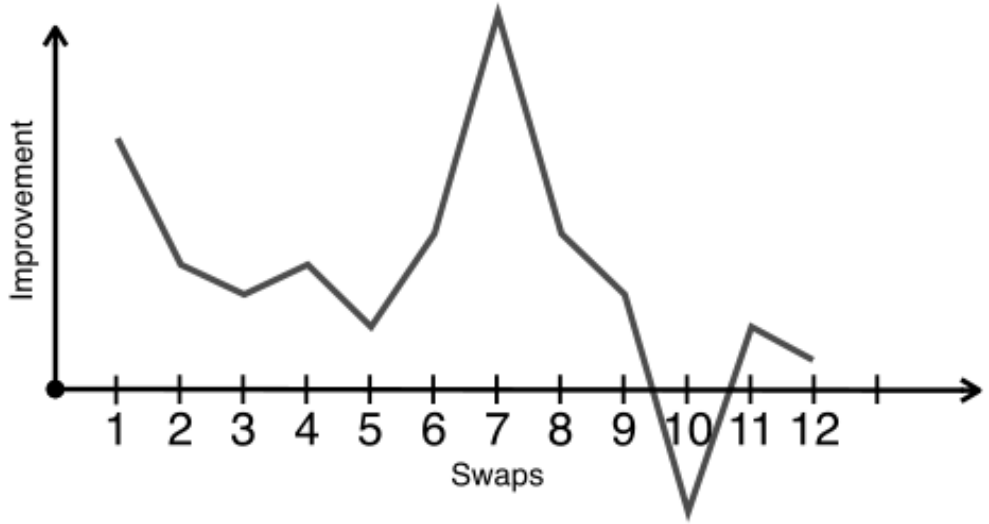
http://akira.ruc.dk/~keld/research/LKH/KoptReport.pdf



Local search for TSP: K-Opt

- Task: find the shortest path to visit all cities exactly once
  - K-OPT (and Lin-Kernighan heuristic improved by Helsgaun - LKH-2):
    - Replace the notion of one favorable swap by a sequence of favorable swaps
    - Do not search for the entire set of sequences but build them incrementally

http://akira.ruc.dk/~keld/research/LKH/KoptReport.pdf



Local search for TSP: K-Opt

- Task: find the shortest path to visit all cities exactly once
  - K-OPT:

    Choose a vertex t_i and an edge (t_i, t_j)

    Choose a vertex t_p with $d(t_j, t_p) < d(t_i, t_j)$

    If none exist, restart

    Consider solution by removing (t_p, t_q) and adding (t_i, t_q)

    Compute the cost but do not connect

    Restart with t_i and edge (t_i, t_q)

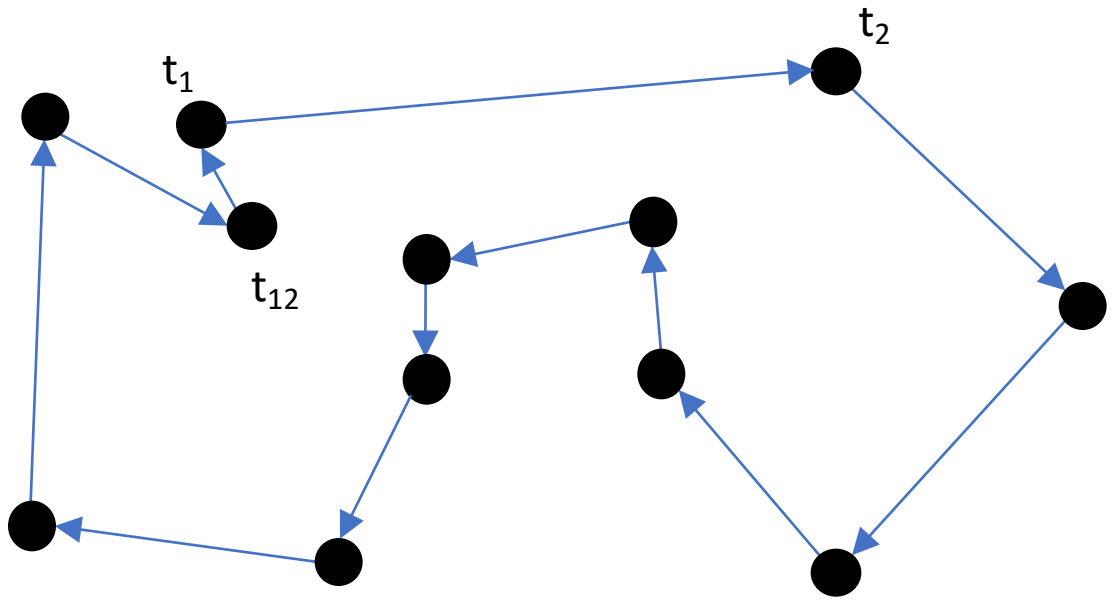
j either predecessor or successor of i

p not connected to j

Delete edge q,p and add edge q,i

Local search for TSP

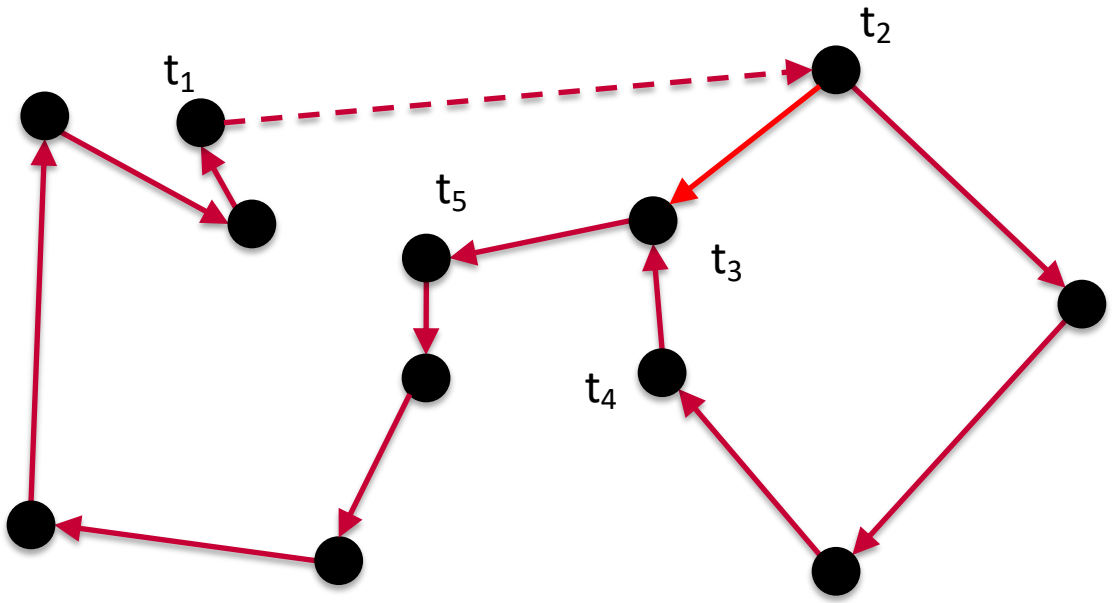
- Task: find the shortest path to visit all cities exactly once
  - K-OPT:



Choose a vertex t_1 and an edge (t_1, t_2)

Local search for TSP

- Task: find the shortest path to visit all cities exactly once
  - K-OPT:

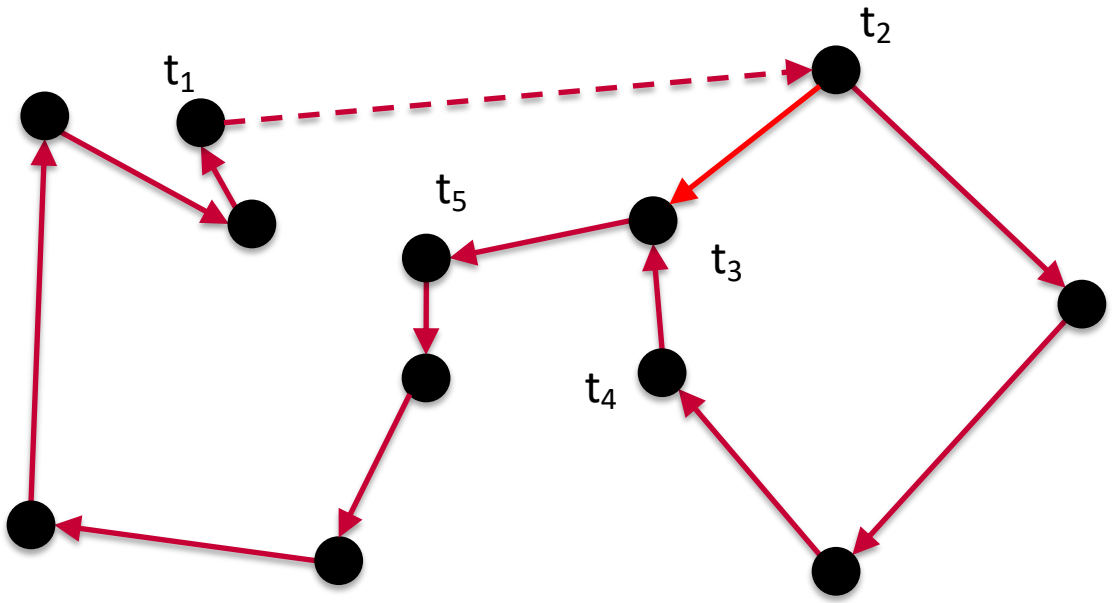


Choose a vertex t_1 and an edge (t_1, t_2)

Choose a vertex t_3 with $d(t_2, t_3) < d(t_1, t_2)$

Local search for TSP

- Task: find the shortest path to visit all cities exactly once
  - K-OPT:



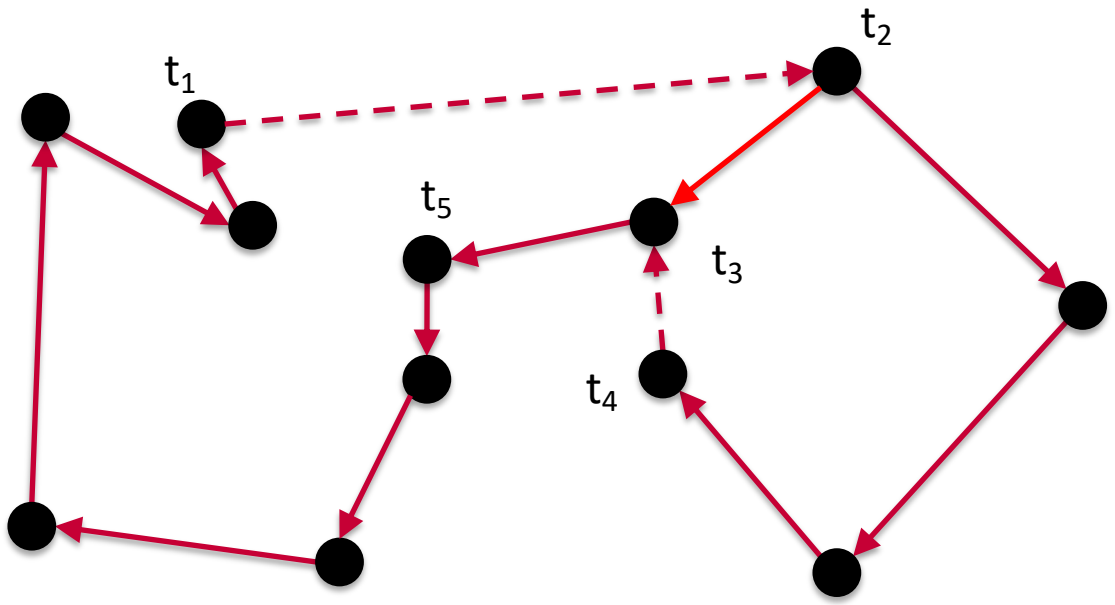
Choose a vertex t_1 and an edge (t_1, t_2)

Choose a vertex t_3 with $d(t_2, t_3) < d(t_1, t_2)$

~~If none exist, restart~~

Local search for TSP

- Task: find the shortest path to visit all cities exactly once
  - K-OPT:



Choose a vertex t_1 and an edge (t_1, t_2)

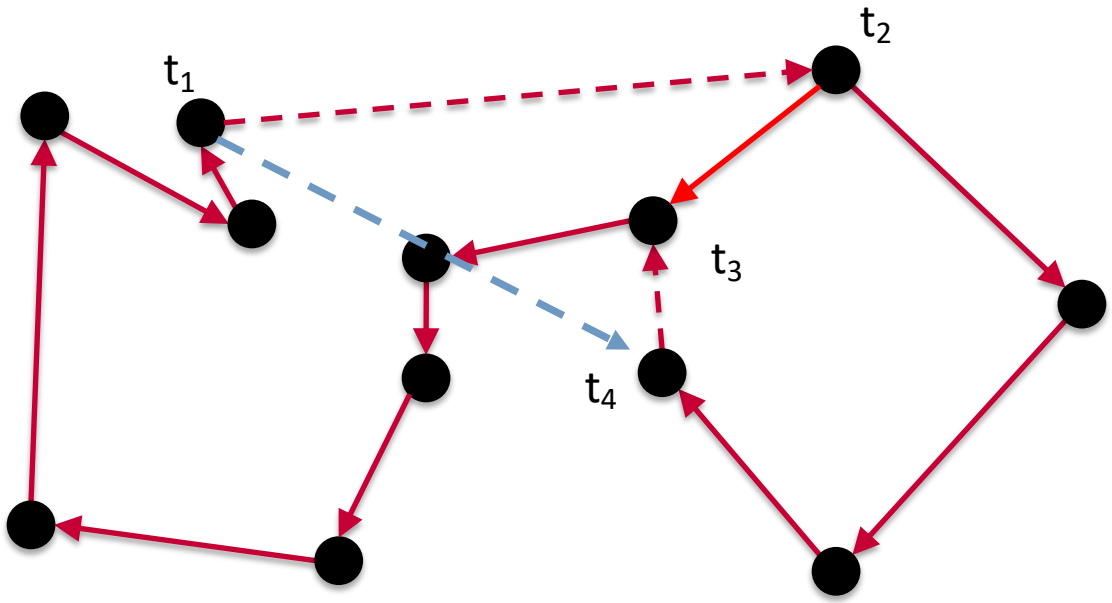
Choose a vertex t_3 with $d(t_2, t_3) < d(t_1, t_2)$

~~If none exist, restart~~

Consider solution by removing (t_3, t_4) and adding (t_1, t_4)

Local search for TSP

- Task: find the shortest path to visit all cities exactly once
  - K-OPT:



Choose a vertex t_1 and an edge (t_1, t_2)

Choose a vertex t_3 with $d(t_2, t_3) < d(t_1, t_2)$

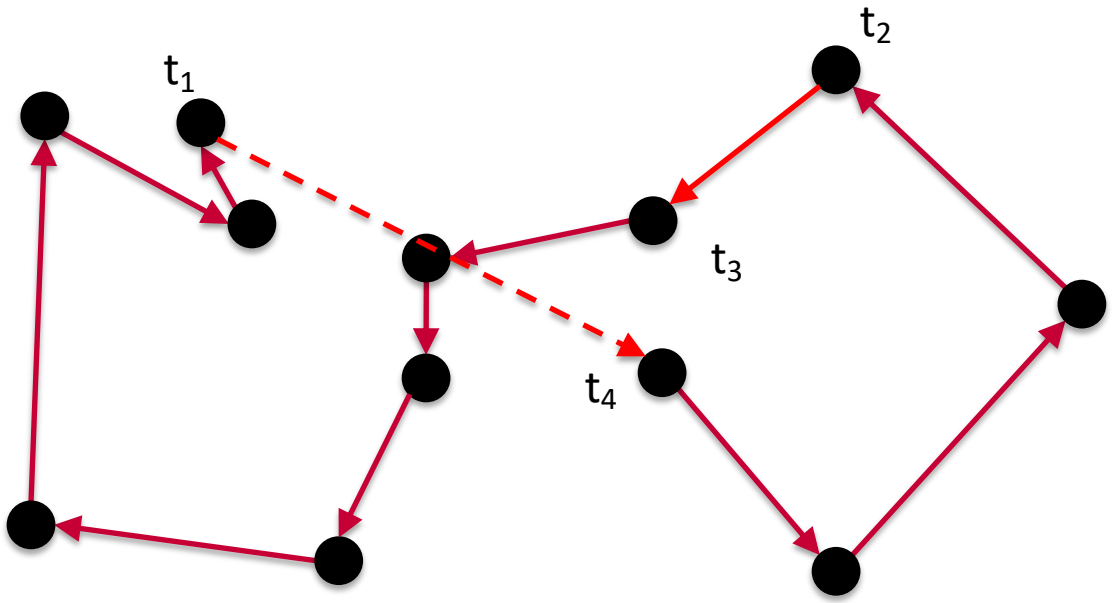
~~If none exist, restart~~

Consider solution by removing (t_3, t_4) and adding (t_1, t_4)

Compute the cost but do not connect

Local search for TSP

- Task: find the shortest path to visit all cities exactly once
  - K-OPT:



Choose a vertex t_1 and an edge (t_1, t_2)

Choose a vertex t_3 with $d(t_2, t_3) < d(t_1, t_2)$

~~If none exist, restart~~

Consider solution by removing (t_3, t_4) and adding (t_1, t_4)

Compute the cost but do not connect

Restart with t_1 and edge (t_1, t_4)

Local search for TSP: K-Opt

- Task: find the shortest path to visit all cities exactly once
  - K-OPT:

    Choose a vertex t_i and an edge (t_i, t_j)

    Choose a vertex t_p with $d(t_j, t_p) < d(t_i, t_j)$

    If none exist, restart

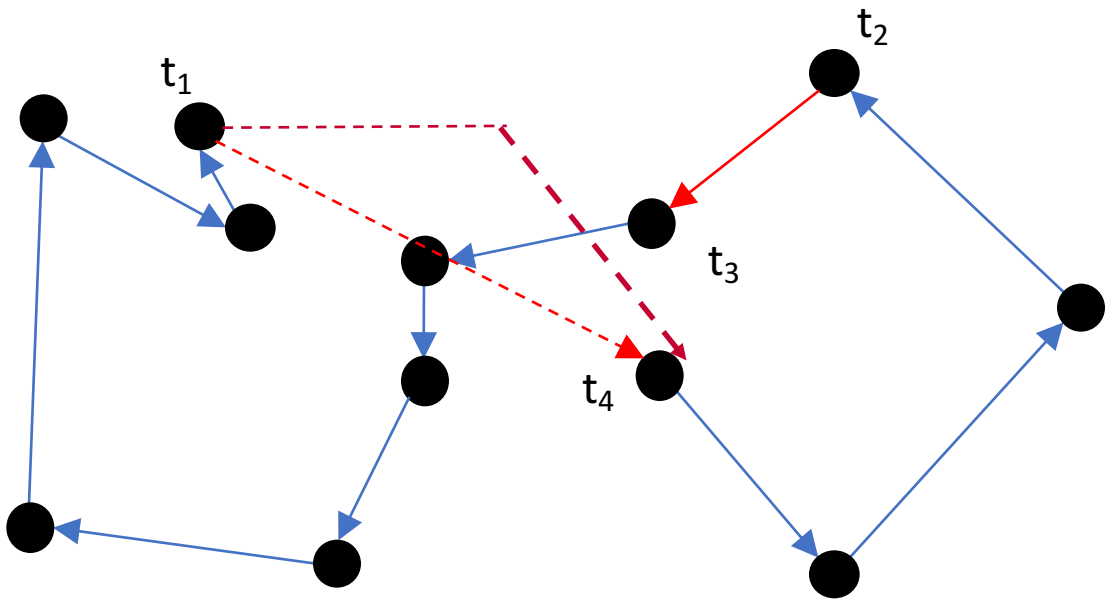
    Consider solution by removing (t_p, t_q) and adding (t_i, t_q)

    Compute the cost but do not connect

    Restart with t_i and edge (t_i, t_q)

Local search for TSP

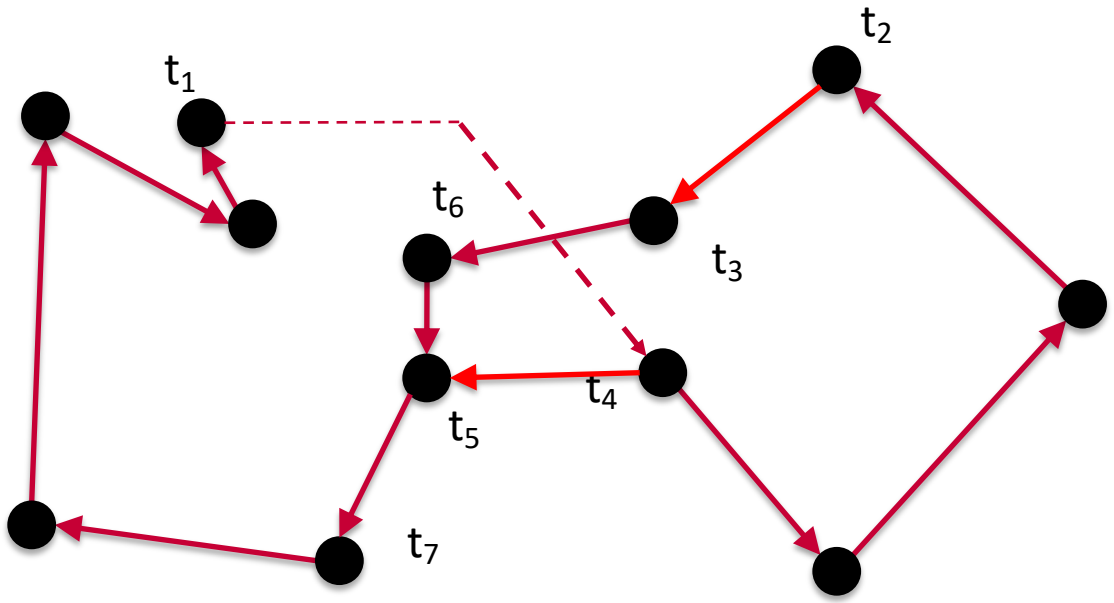
- Task: find the shortest path to visit all cities exactly once
  - K-OPT:



Choose a vertex t_1 and an edge (t_1, t_4)

Local search for TSP

- Task: find the shortest path to visit all cities exactly once
  - K-OPT:

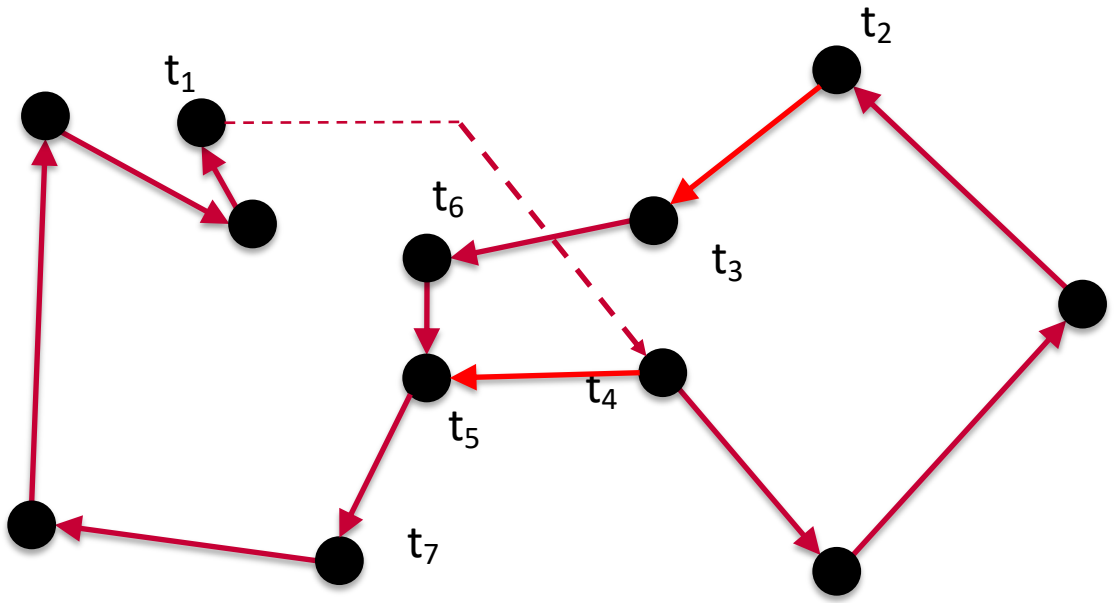


Choose a vertex t_1 and an edge (t_1, t_4)

Choose a vertex t_5 with $d(t_4, t_5) < d(t_1, t_4)$

Local search for TSP

- Task: find the shortest path to visit all cities exactly once
  - K-OPT:



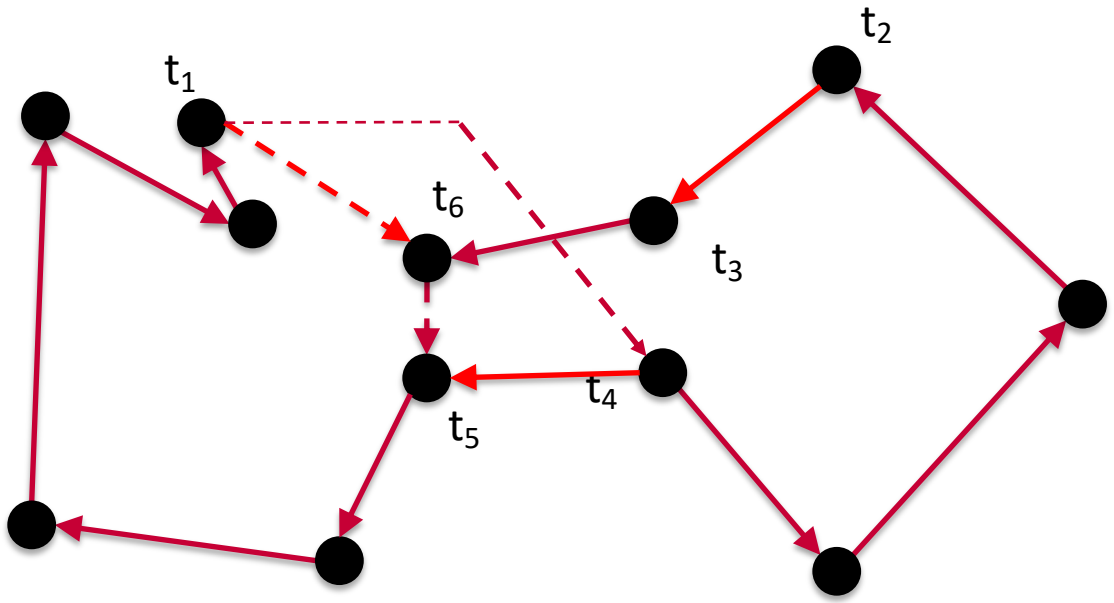
Choose a vertex t_1 and an edge (t_1, t_4)

Choose a vertex t_5 with $d(t_4, t_5) < d(t_1, t_4)$

~~If none exist, restart~~

Local search for TSP

- Task: find the shortest path to visit all cities exactly once
  - K-OPT:



Choose a vertex t_1 and an edge (t_1, t_4)

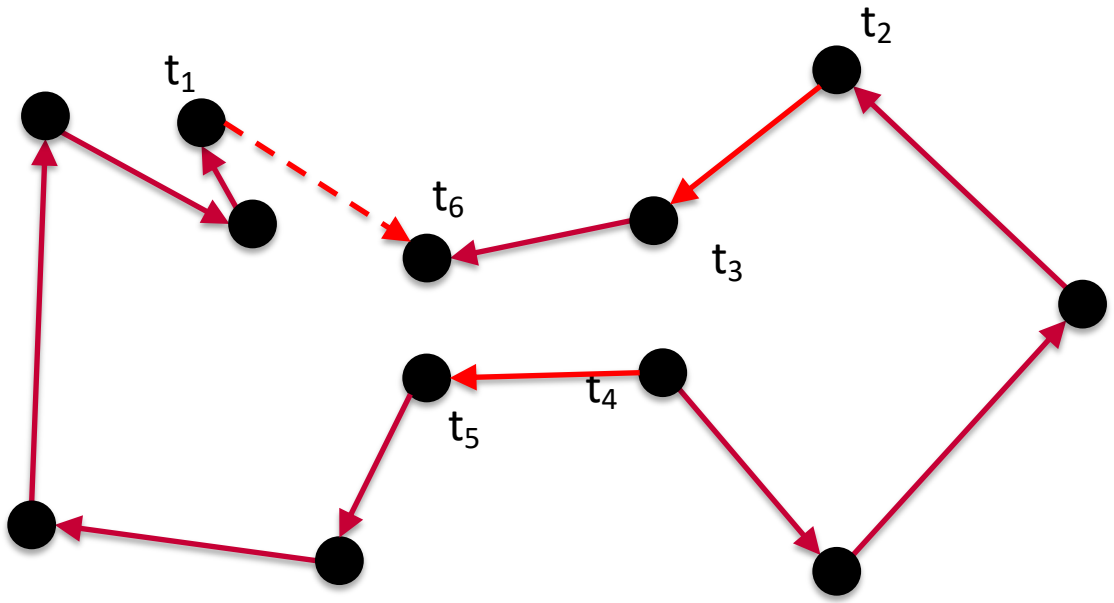
Choose a vertex t_5 with $d(t_4, t_5) < d(t_1, t_4)$

~~If none exist, restart~~

Consider solution by removing (t_5, t_6) and adding (t_1, t_6)

Local search for TSP

- Task: find the shortest path to visit all cities exactly once
  - K-OPT:



Choose a vertex t_1 and an edge (t_1, t_4)

Choose a vertex t_5 with $d(t_4, t_5) < d(t_1, t_4)$

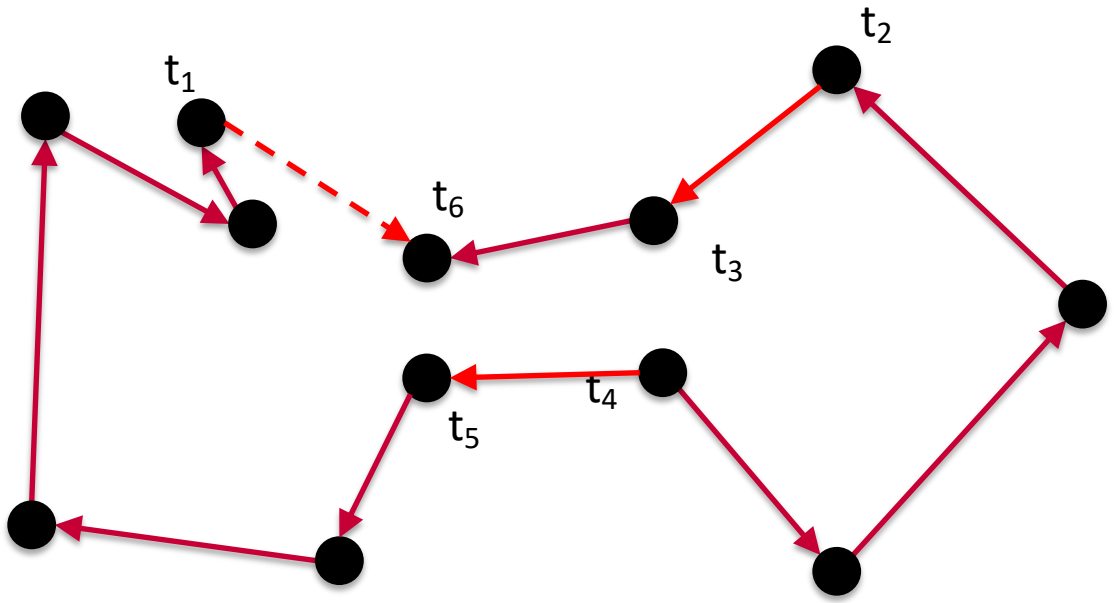
~~If none exist, restart~~

Consider solution by removing (t_5, t_6) and adding (t_1, t_6)

Compute the cost but do not connect

Local search for TSP

- Task: find the shortest path to visit all cities exactly once
  - K-OPT:



Choose a vertex t_1 and an edge (t_1, t_4)

Choose a vertex t_5 with $d(t_4, t_5) < d(t_1, t_4)$

~~If none exist, restart~~

Consider solution by removing (t_5, t_6) and adding (t_1, t_6)

Compute the cost but do not connect

Restart with t_1 and edge (t_1, t_6)

Local search for TSP: K-Opt

- Task: find the shortest path to visit all cities exactly once
  - K-OPT:

    Choose a vertex t_i and an edge (t_i, t_j)

    Choose a vertex t_p with $d(t_j, t_p) < d(t_i, t_j)$

    If none exist, restart

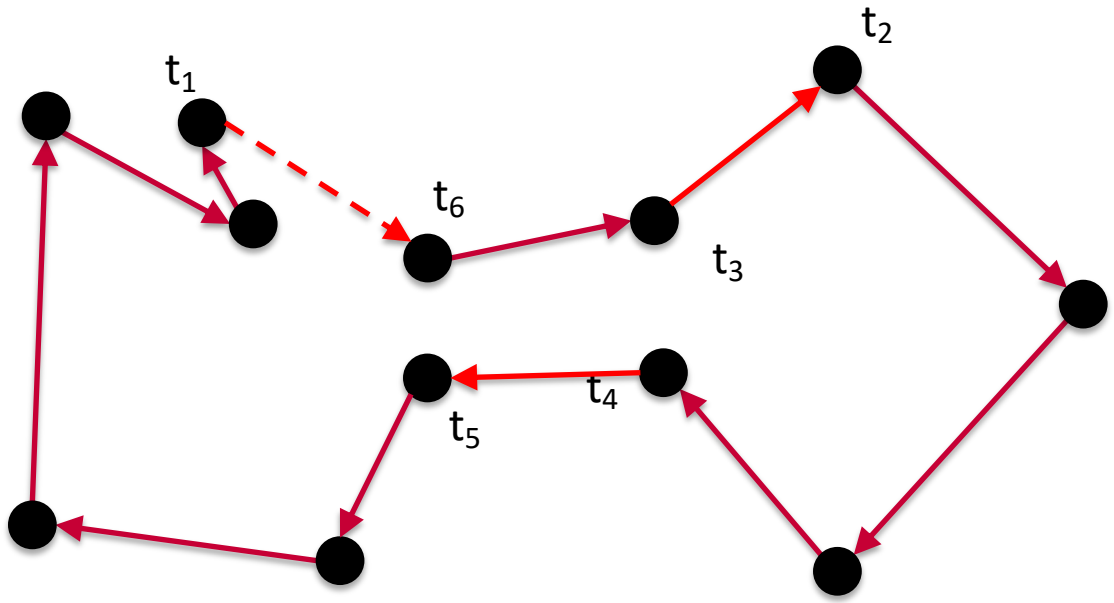
    Consider solution by removing (t_p, t_q) and adding (t_i, t_q)

    Compute the cost but do not connect

    Restart with t_i and edge (t_i, t_q)

Local search for TSP

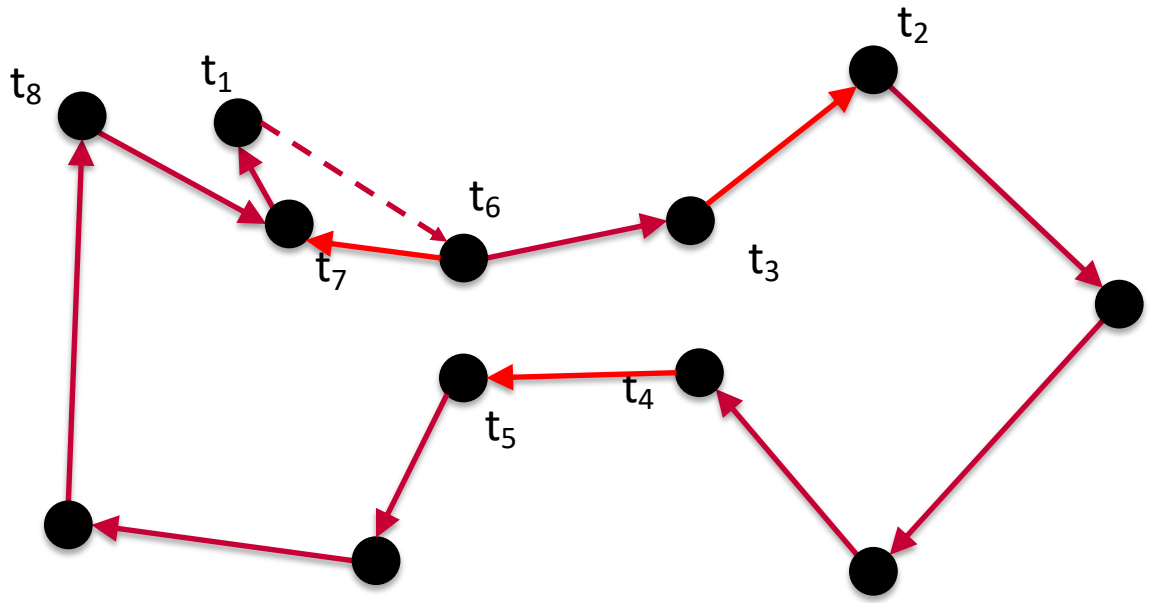
- Task: find the shortest path to visit all cities exactly once
  - K-OPT:



Choose a vertex t_1 and an edge (t_1, t_6)

Local search for TSP

- Task: find the shortest path to visit all cities exactly once
  - K-OPT:

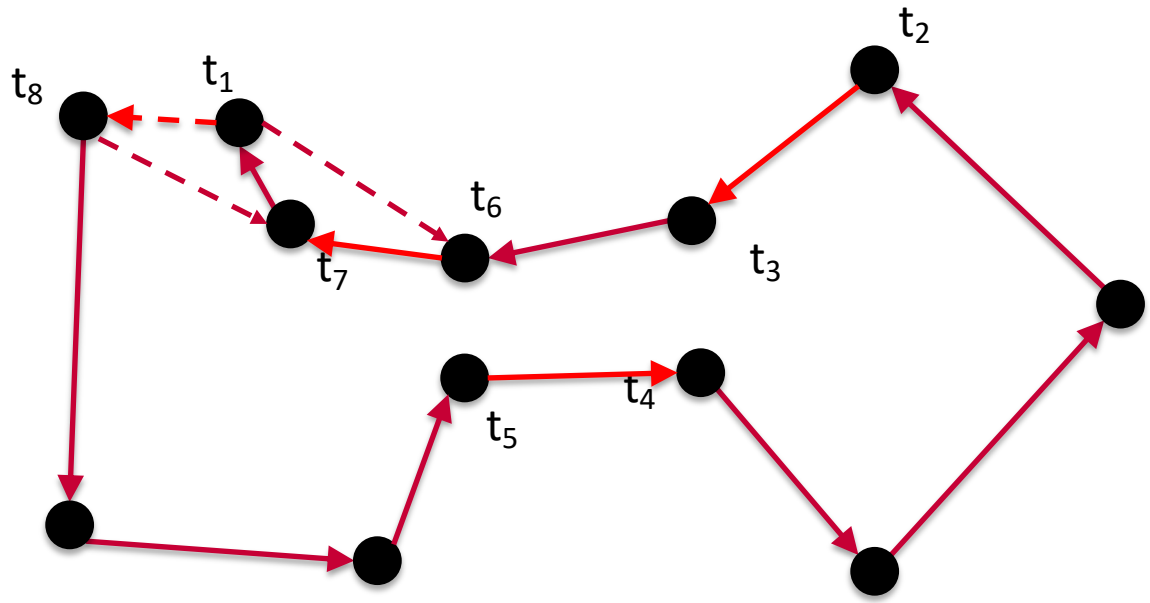


Choose a vertex t_1 and an edge (t_1, t_6)

Choose a vertex t_7 with $d(t_7, t_6) < d(t_1, t_6)$

Local search for TSP

- Task: find the shortest path to visit all cities exactly once
  - K-OPT:



Choose a vertex t_1 and an edge (t_1, t_6)

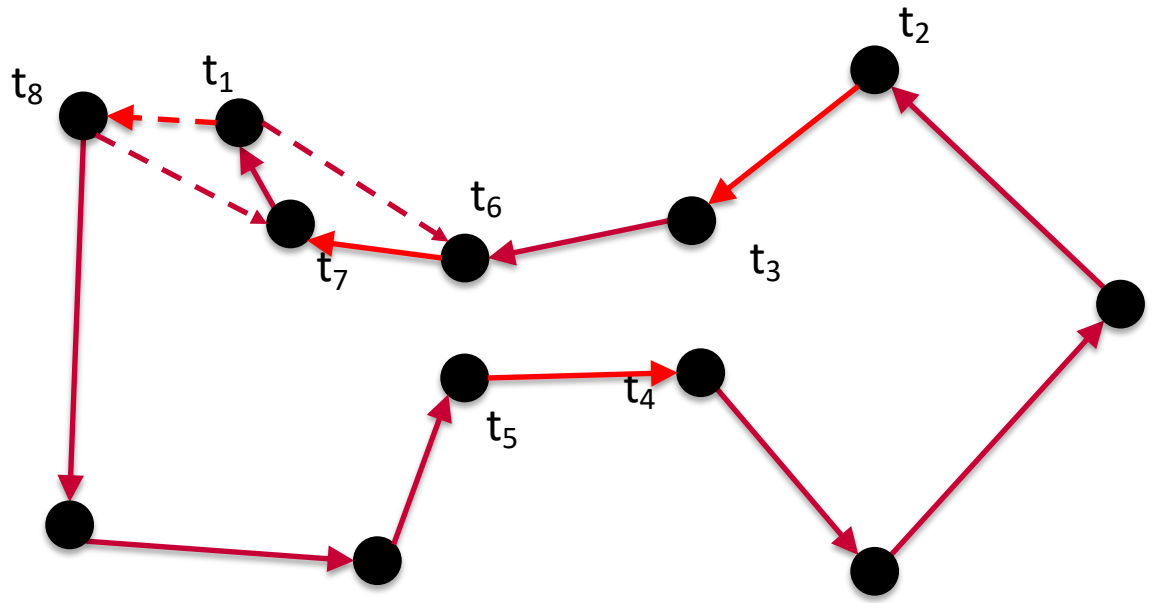
Choose a vertex t_5 with $d(t_7, t_6) < d(t_1, t_6)$

~~If none exist, restart~~

Consider solution by removing (t_8, t_7) and adding (t_1, t_8)

Local search for TSP

- Task: find the shortest path to visit all cities exactly once
  - K-OPT:



Choose a vertex t_1 and an edge (t_1, t_6)

Choose a vertex t_5 with $d(t_7, t_6) < d(t_1, t_6)$

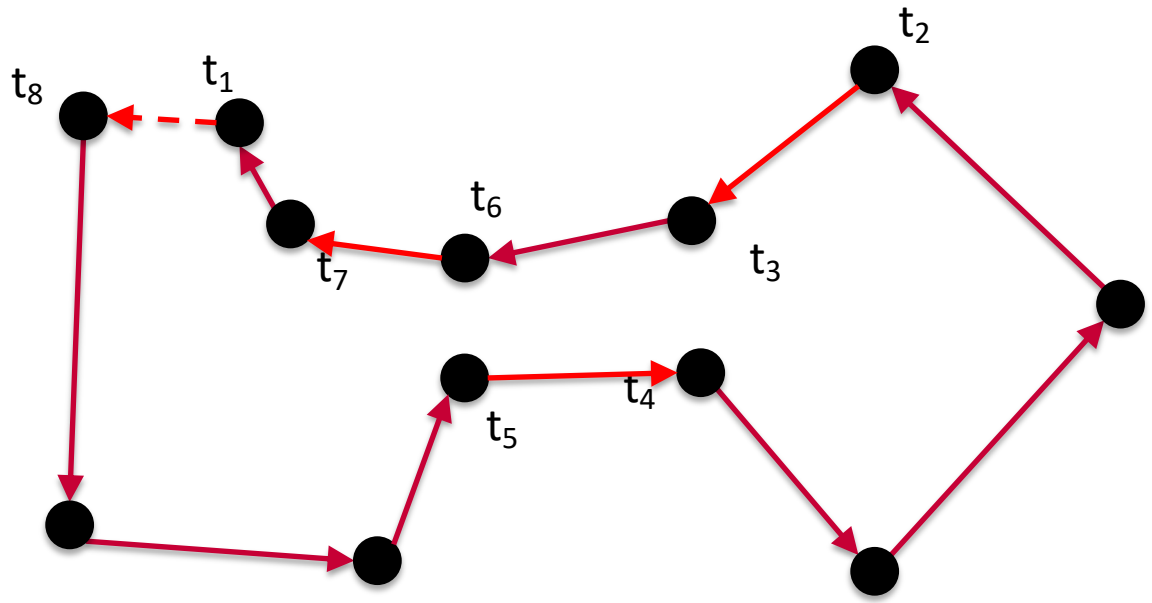
~~If none exist, restart~~

Consider solution by removing (t_8, t_7) and adding (t_1, t_8)

Compute the cost but do not connect

Local search for TSP

- Task: find the shortest path to visit all cities exactly once
  - K-OPT:



Choose a vertex t_1 and an edge (t_1, t_6)

Choose a vertex t_5 with $d(t_1, t_8) < d(t_8, t_7)$

~~If none exist, restart~~

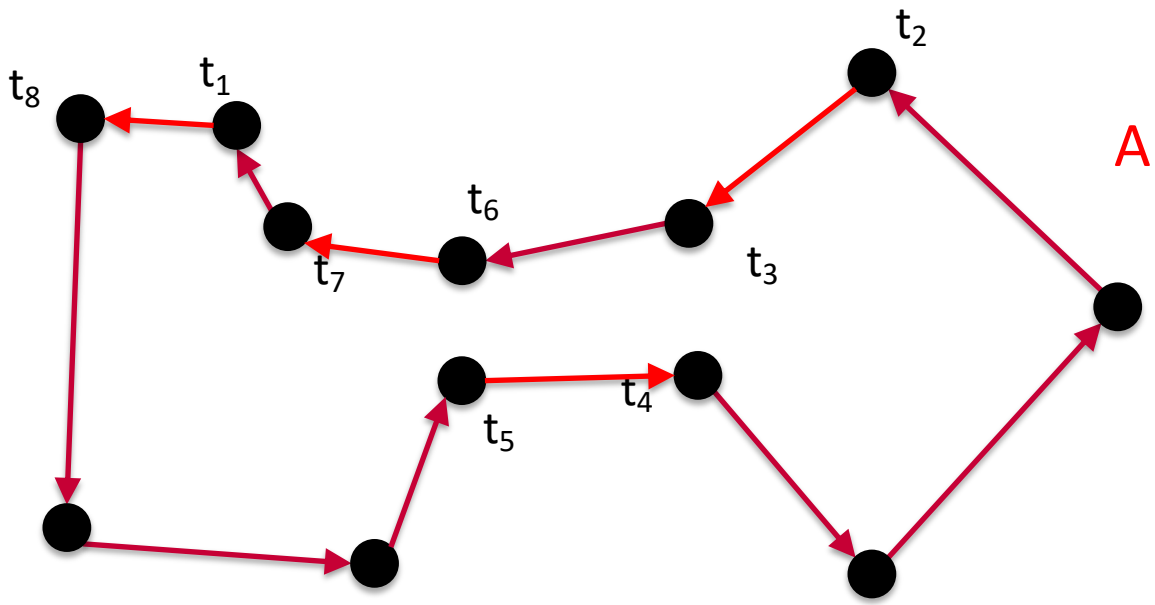
Consider solution by removing (t_8, t_7) and adding (t_1, t_8)

Compute the cost but do not connect

Restart with t_1 and edge (t_1, t_8)

Local search for TSP

- Task: find the shortest path to visit all cities exactly once
  - K-OPT:



A good move!

Local search for TSP

- Description from http://akira.ruc.dk/~keld/research/LKH/KoptReport.pdf

```
for (each t[2] in {PRED(t[1]), SUC(t[1])}) {
    G[0] = C(t[1], t[2]);
    for (each candidate edge (t[2], t[3])) {
        if (t[3] != PRED(t[2]) && t[3] != SUC(t[2]) &&
            (G[1] = G[0] - C(t[2], t[3]) > 0) {
            for (each t[4] in {PRED(t[3]), SUC(t[3])}) {
                G[2] = G[1] + C(t[3], t[4]);
                if (FeasibleKOptMove(2) &&
                    (Gain = G[2] - C(t[4], t[1])) > 0) {
                    MakeKOptMove(2);
                    return Gain;
                }
            }
            inner loops for choosing t[5], ..., t[2K]
        }
    }
}
```

predecessor / successor

"Gain" variable

t3 != pred/succ of t2

And +ve Gain?

t4 s.t. feasible to move to t1 and gain stays +ve

Local search for TSP

- Description of recursive fn from http://akira.ruc.dk/~keld/research/LKH/KoptReport.pdf

Added/Deleted make sure we don't add/delete an edge from earlier

Closing of all loops unseen

```
GainType BestKOptMoveRec(int k, GainType G0) {
    GainType G1, G2, G3, Gain;
    Node *t1 = t[1], *t2 = t[2 * k - 2], *t3, *t4;
    int i;

    for (each candidate edge (t2, t3)) {
        if (t3 != PRED(t2) && t3 != SUC(t2) &&
            !Added(t2, t3, k - 2) &&
            (G1 = G0 - C(t2, t3)) > 0) {
            t[2 * k - 1] = t3;
            for (each t4 in {PRED(t3), SUC(t3)}) {
                if (!Deleted(t3, t4, k - 2)) {
                    t[2 * k] = t4;
                    G2 = G1 + C(t3, t4);
                    if (FeasibleKOptMove(k) &&
                        (Gain = G2 - C(t4, t1)) > 0) {
                        MakeKOptMove(k);
                        return Gain;
                    }
                    if (k < K &&
                        (Gain = BestKOptMoveRec(k + 1, G2)) > 0)
                        return Gain;
                    if (k == K && G2 > BestG2K &&
                        Excludable(t3, t4)) {
                        BestG2K = G2;
                        for (i = 1; i <= 2 * K; i++)
                            Best_t[i] = t[i];
                    }
```

Constructive heuristics for TSP

- ‘Simple’ LS algorithms that quickly construct reasonable good tours
- These are often used to provide an initial search position for more advanced LS algorithms
- Various types of constructive search algorithms exist
  - Iteratively extend a connected partial tour
  - Iteratively build tour fragments and patch them together into a complete tour
  - Algorithms based on minimum spanning trees

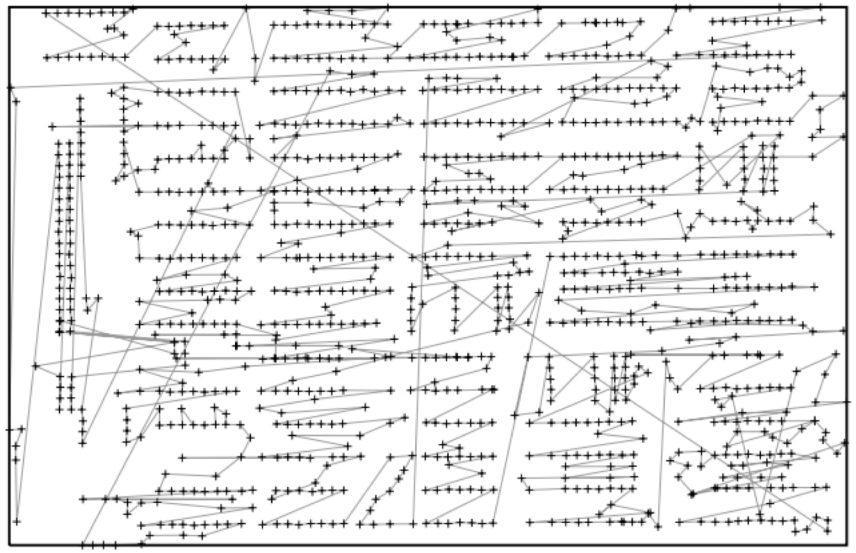
Nearest neighbor (NN) construction heuristics:

- Start with a single vertex (chosen uniformly at random)
- In each step, follow minimal-weight edge to yet unvisited next vertex
- Complete Hamiltonian cycle by adding initial vertex to the end of the path
- Results on length of NN tours
  - For TSP instances with triangle inequality NN is at most 0.5*(log2(n) + 1) worse than optimal solution

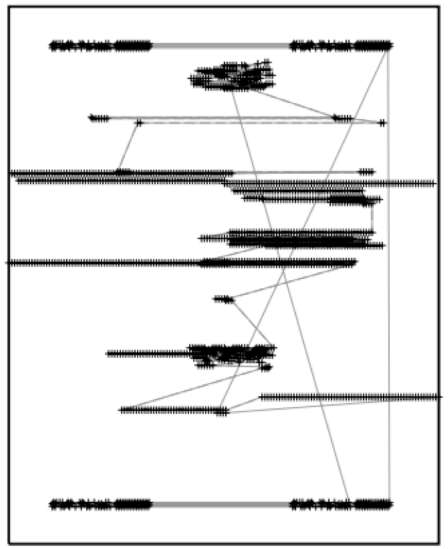
Two examples of NN tour for TSPLIB instances

- For metric and TSPLIB instances, NN tours are typically within 20 - 30% of optimal
- Typically, NN tours are locally close to optimal, but contain few long edges

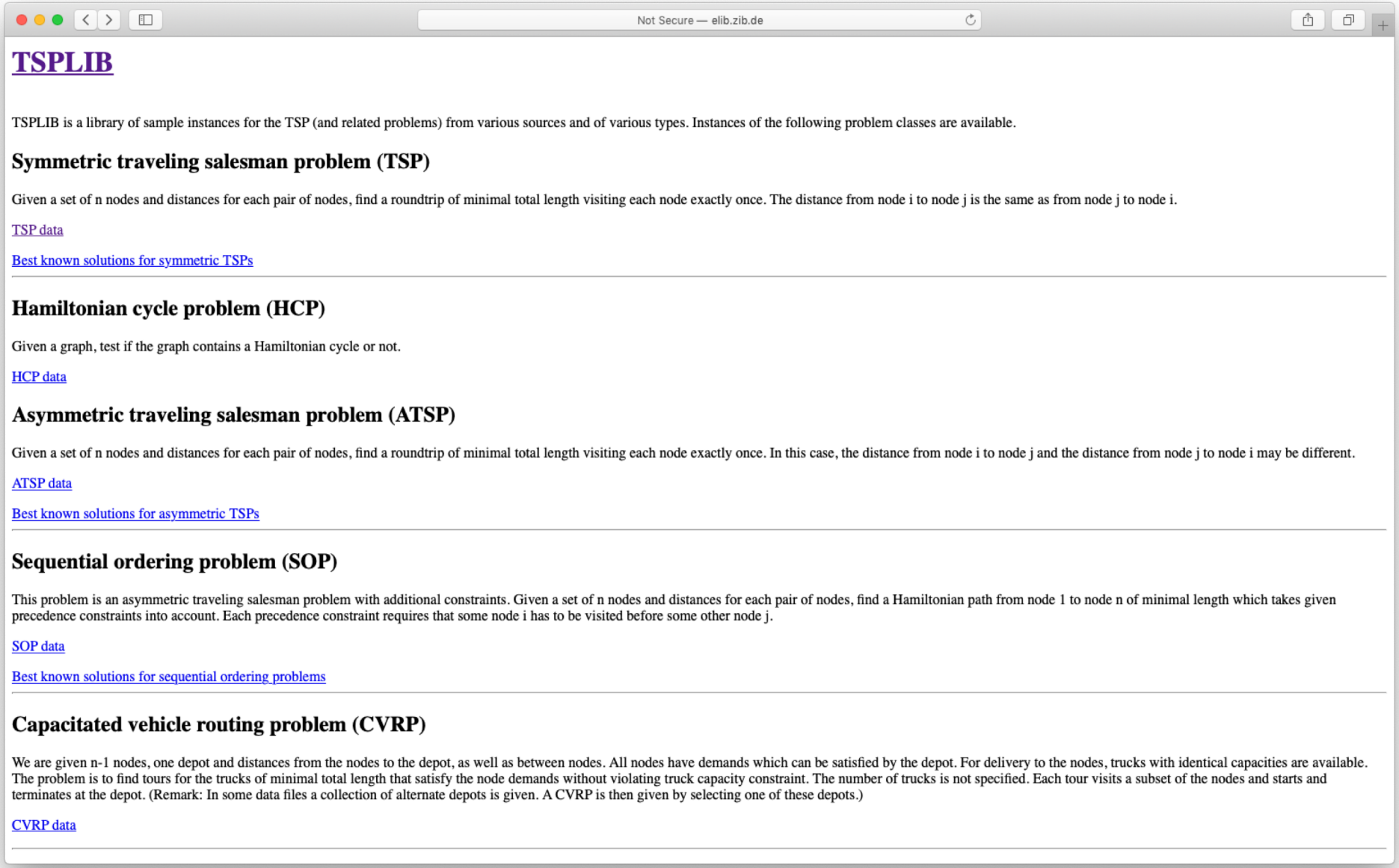
pcb1173



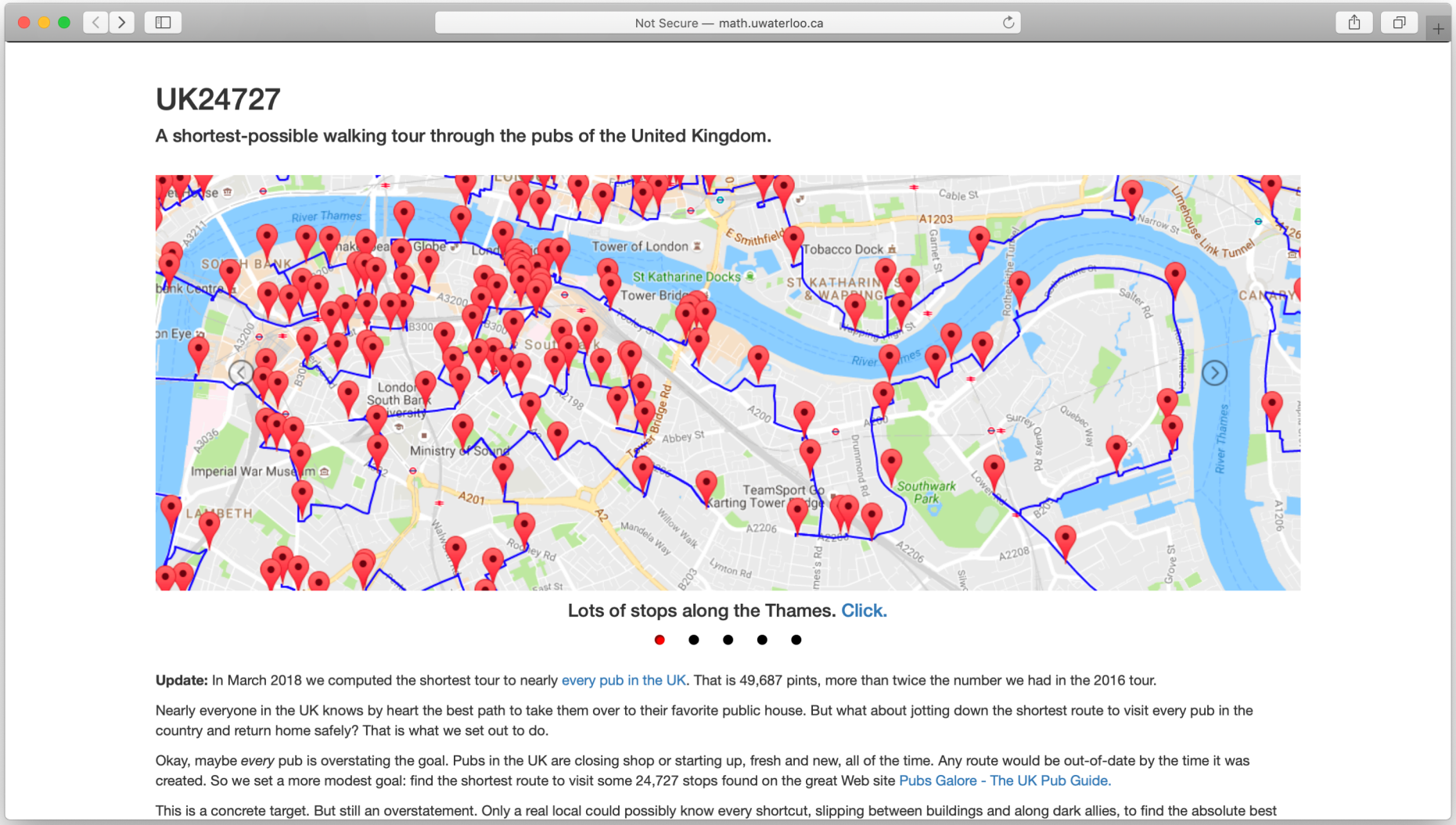
fl1577



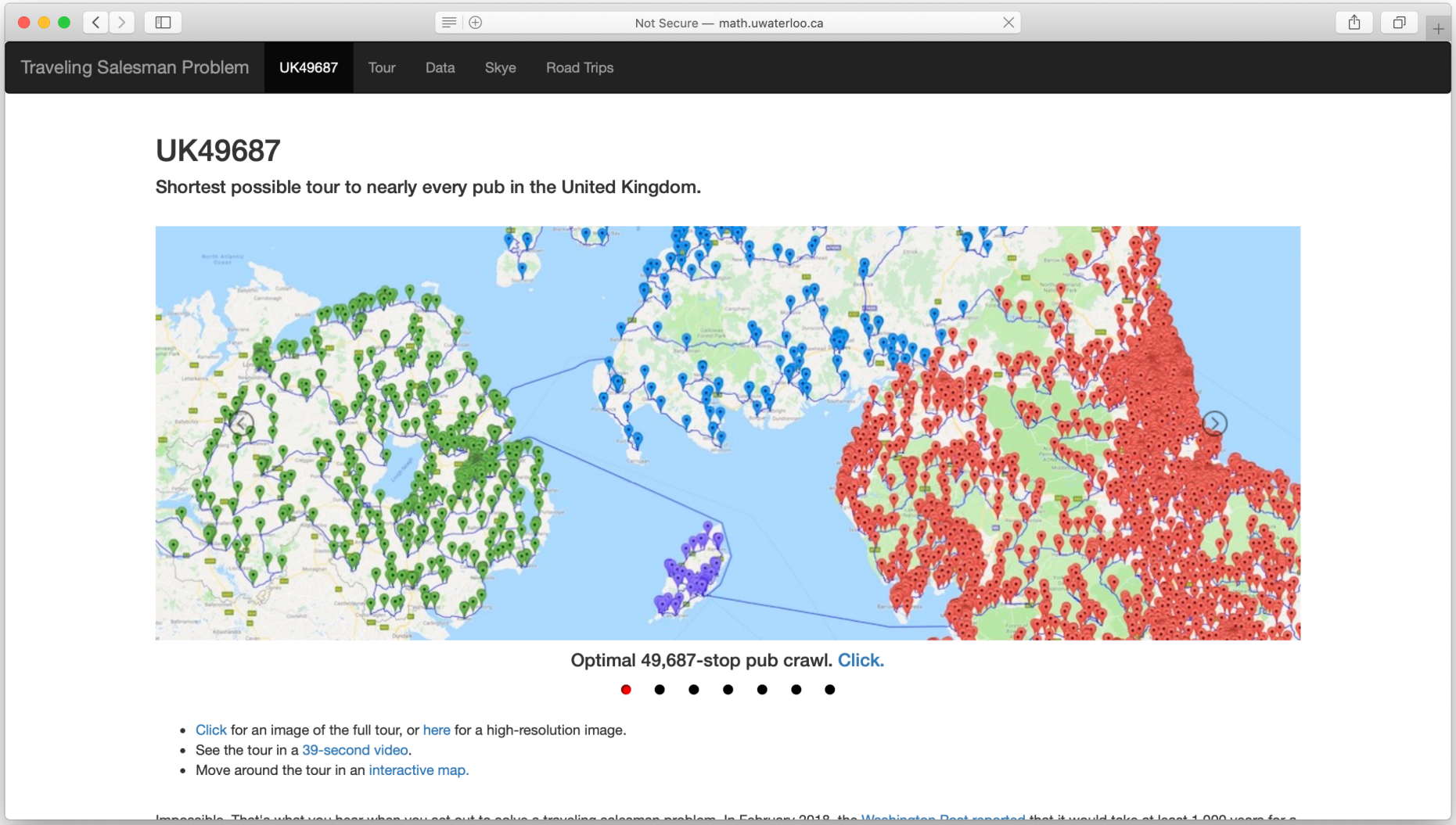
http://elib.zib.de/pub/mp-testdata/tsp/tsplib/tsplib.html



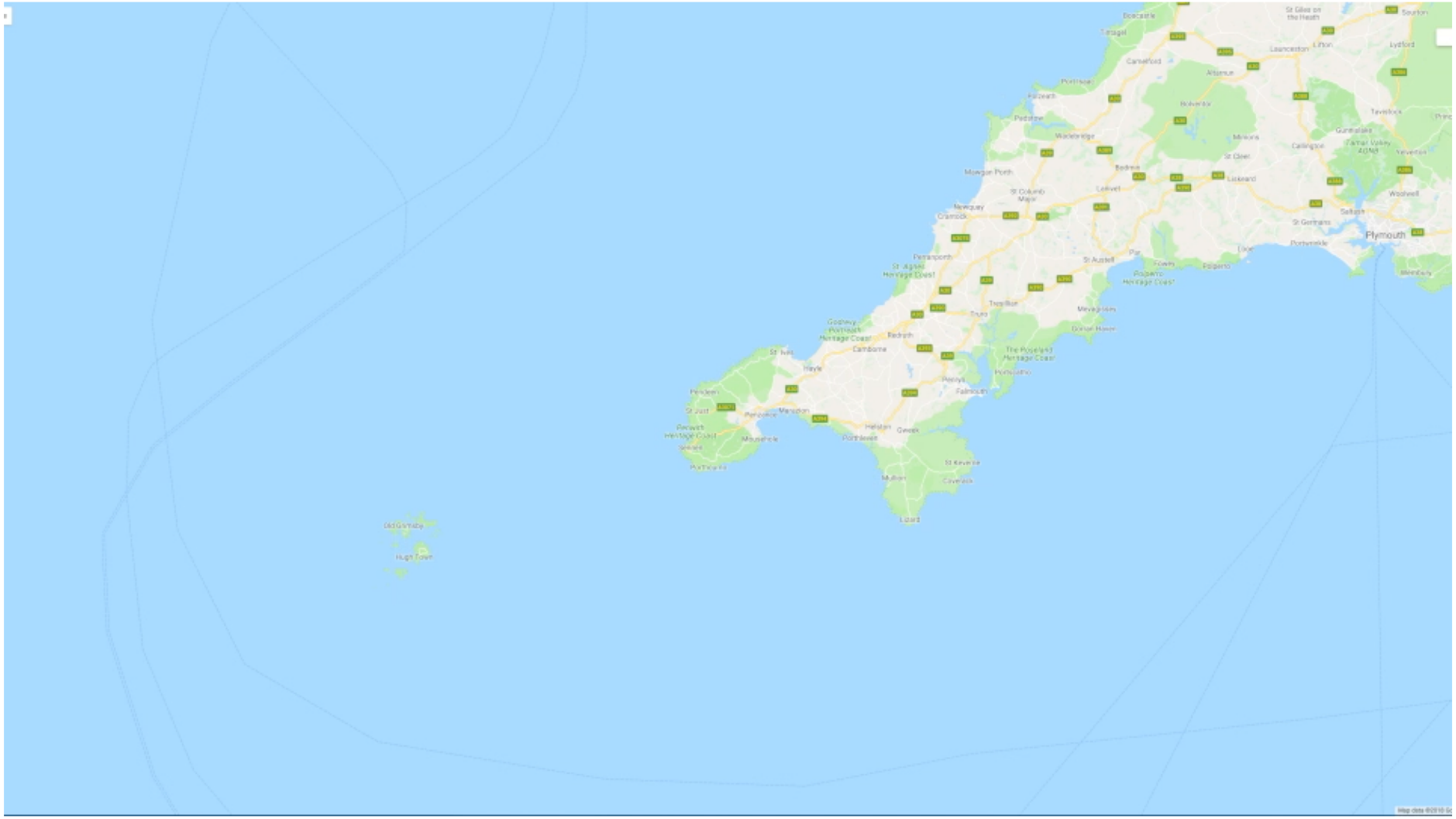
http://www.math.uwaterloo.ca/tsp/pubs/



http://www.math.uwaterloo.ca/tsp/uk/index.html



http://www.math.uwaterloo.ca/tsp/uk/uk49_run720.mp4



Insertion heuristics

- Insertion heuristics iteratively extend a partial tour p by inserting a heuristically chosen vertex such that the path length increases minimally
- Various heuristics for the choice of the next vertex to insert.
- Nearest cheap insertion guarantee approximation ratio of two for TSP instances with triangle inequality
- In practice, farthest and random insertion perform better; typically, 13-15% above optimal for metric and TSPLIB instances

Tabu Search

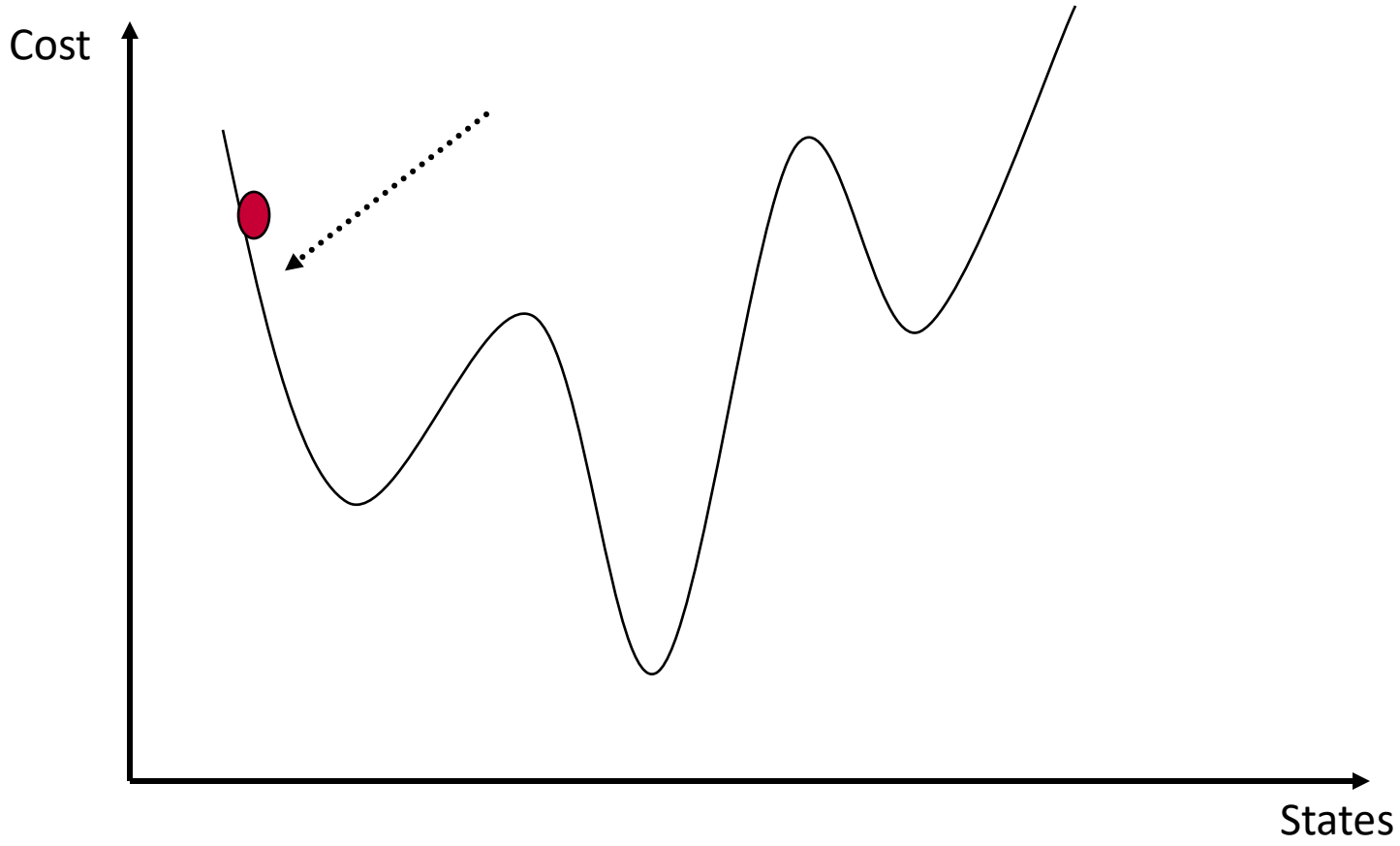
Tabu Search

- *“Tabu search is a "higher level" heuristic procedure for solving optimization problems, designed to guide other methods (or their component processes) to escape the trap of local optima”*

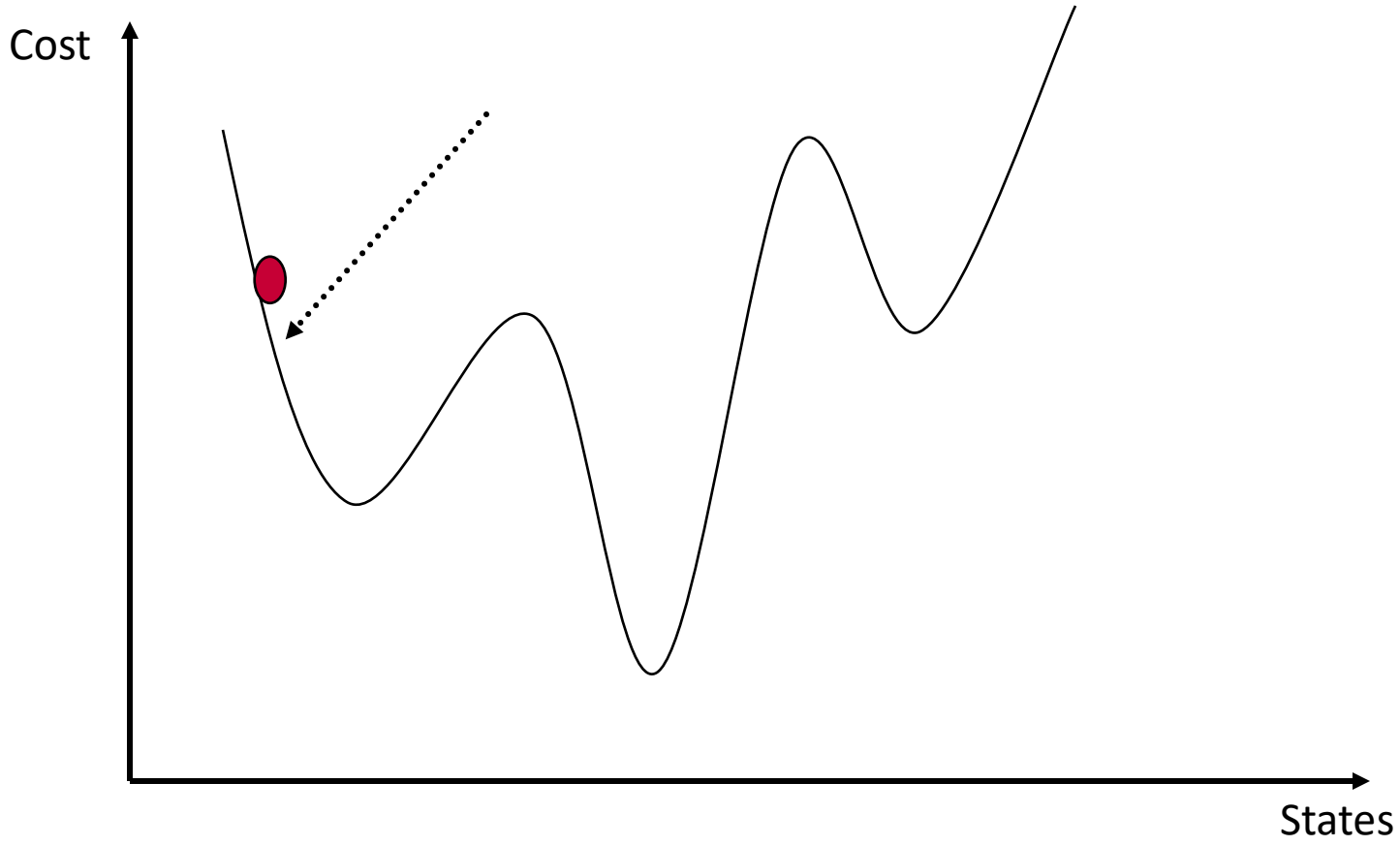
- Local search can get stuck in loops bouncing between the same set of states, since no memory!

- Tabu search: Allow worsening moves but prevent returning quickly to the same state
  - Keep fixed length queue “Tabu list”
  - Add most recent state to queue; drop oldest
  - Never go to a state that is currently tabu’ed

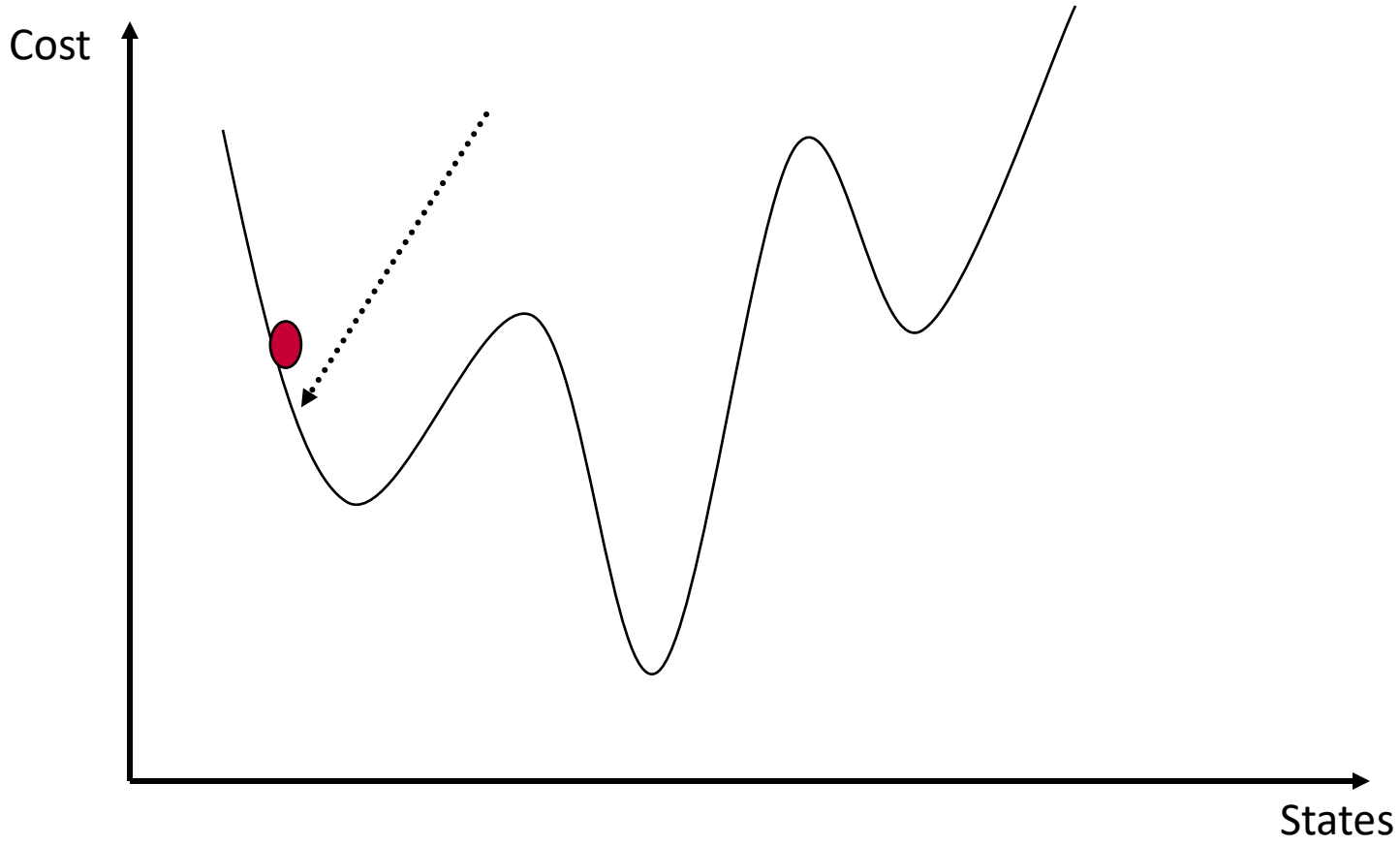
Greedy hill climbing (minimization)



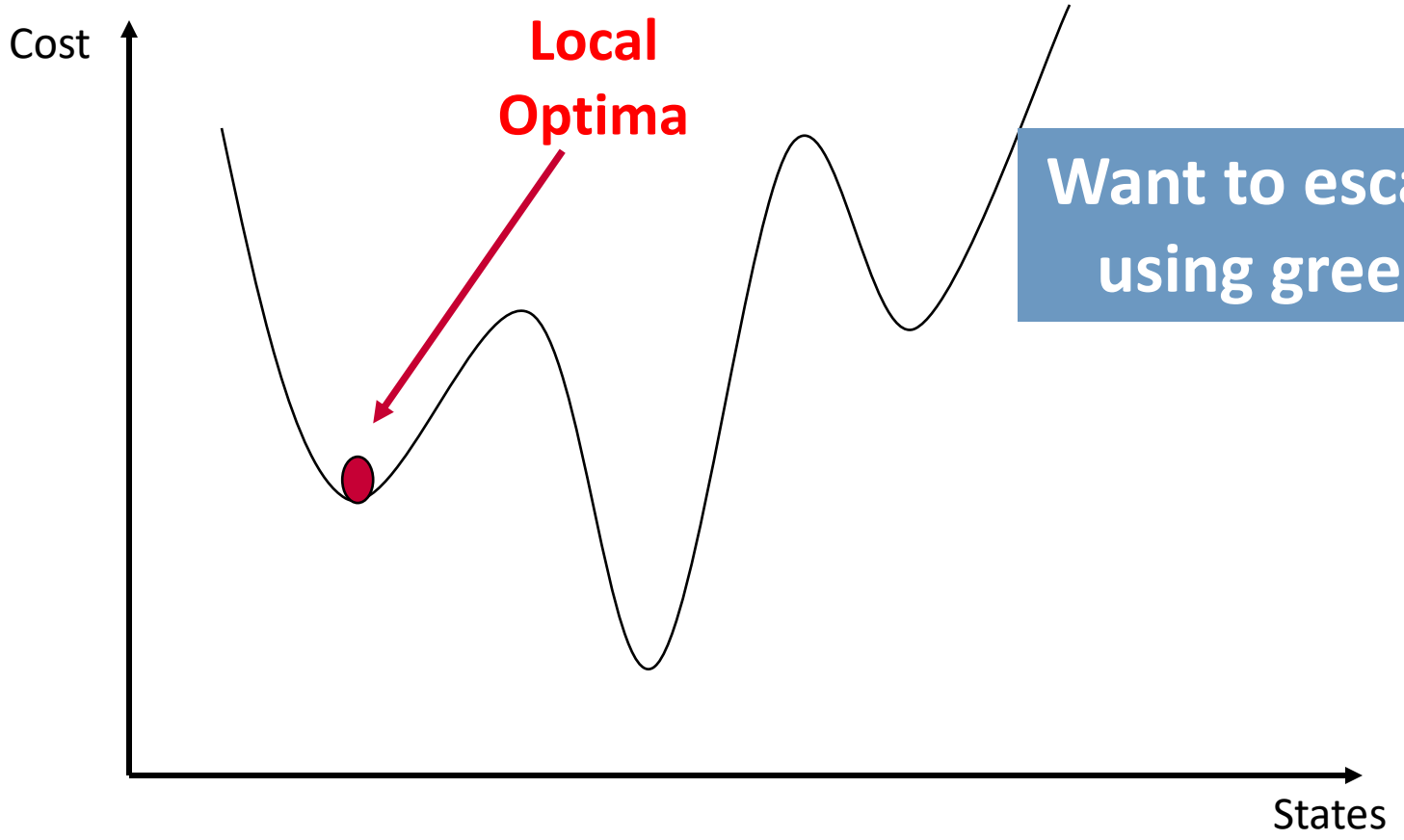
Greedy hill climbing (minimization)



Greedy hill climbing (minimization)

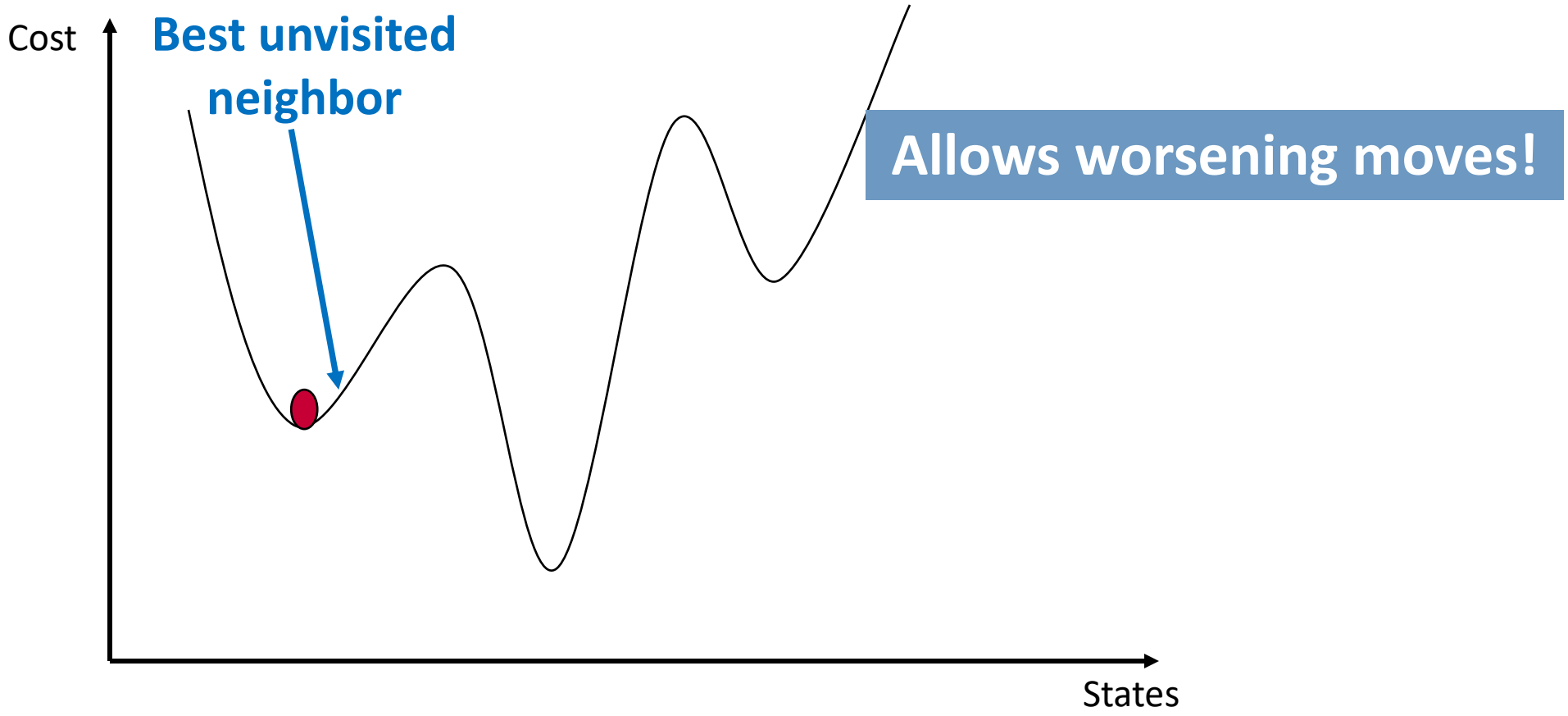


Greedy hill climbing (minimization)

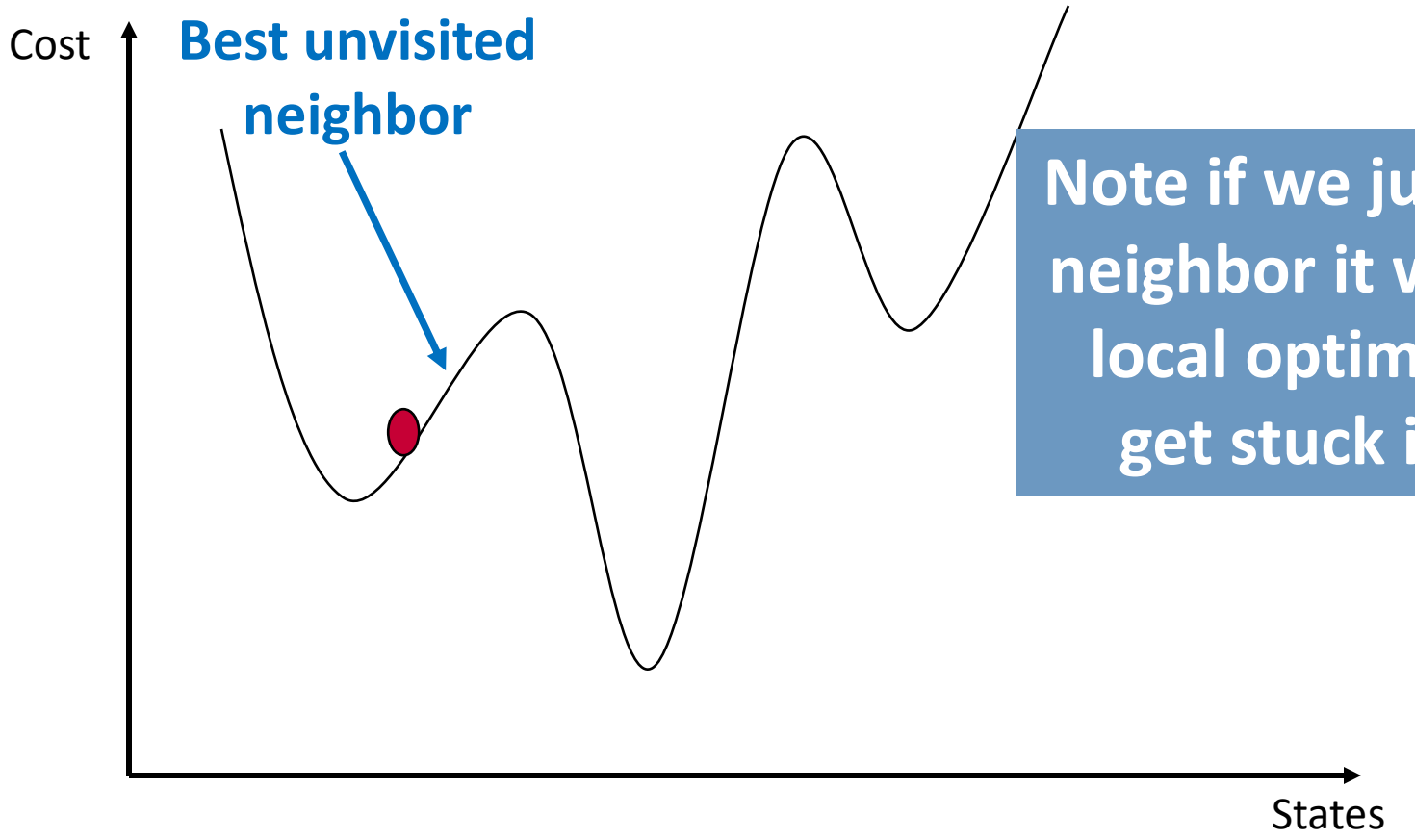


Want to escape while still using greedy selection

Tabu Search + Greedy hill climbing (minimization)

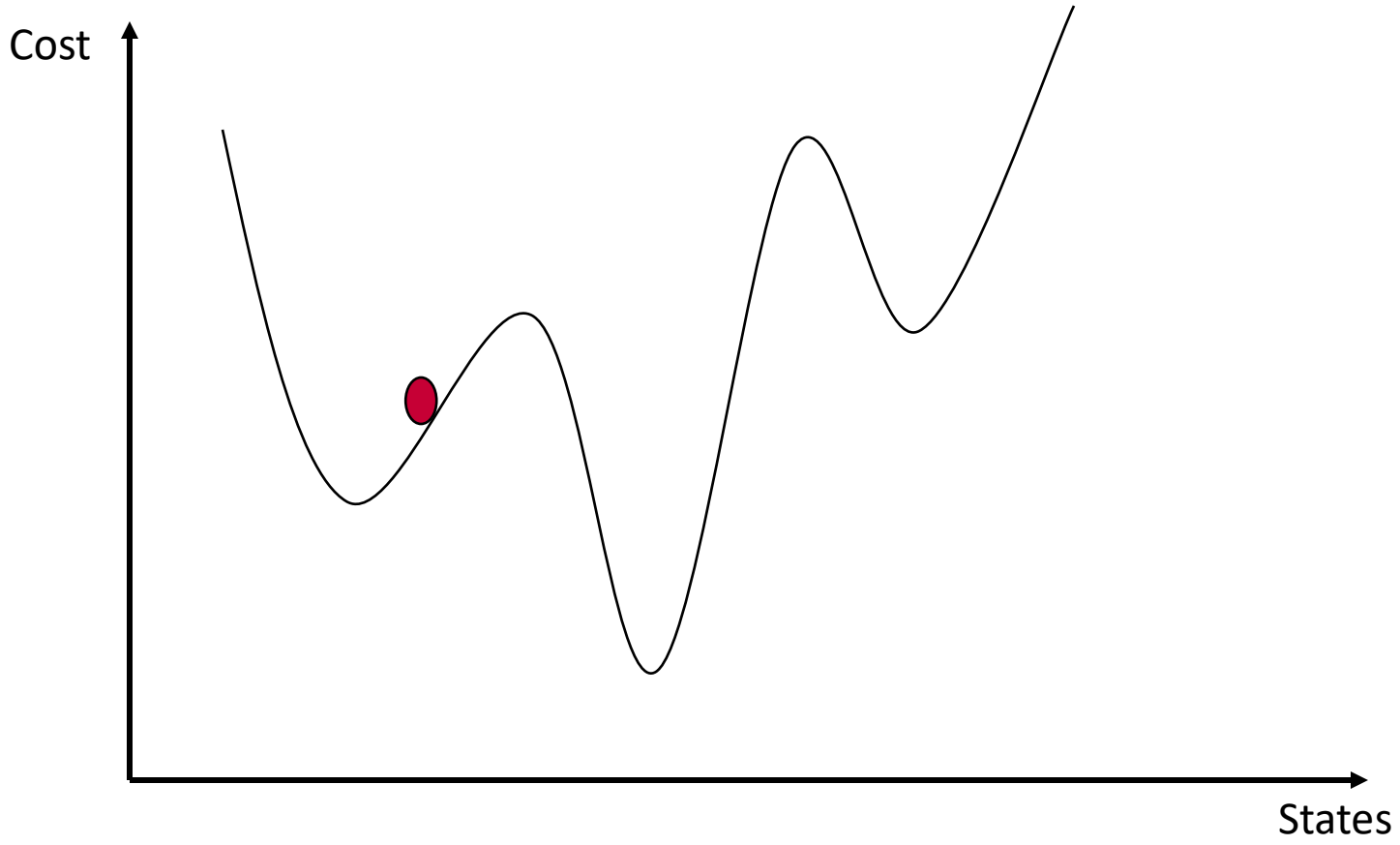


Tabu Search + Greedy hill climbing (minimization)

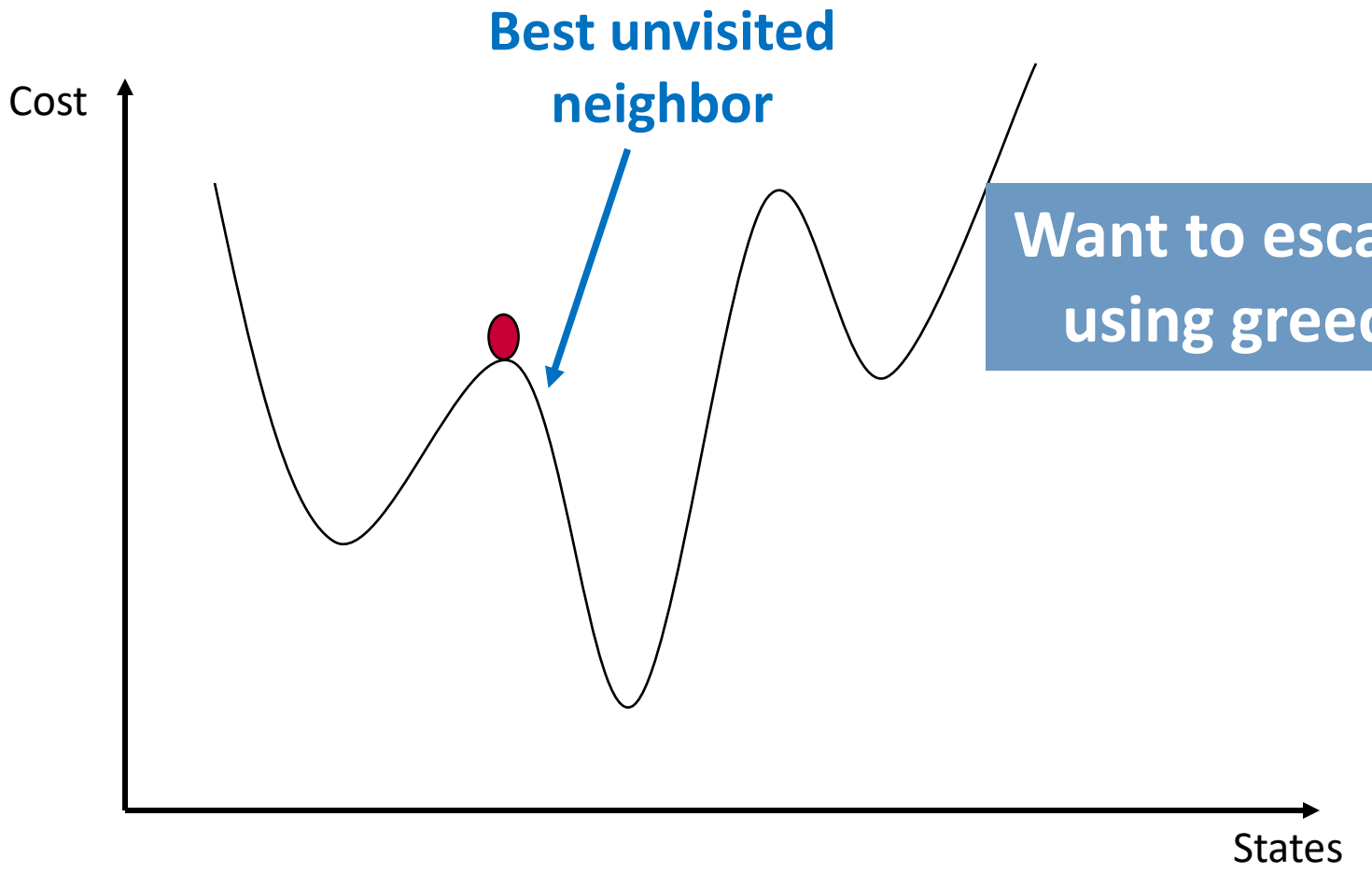


Note if we just chose best neighbor it would choose local optima again and get stuck in this loop

Tabu Search + Greedy hill climbing (minimization)

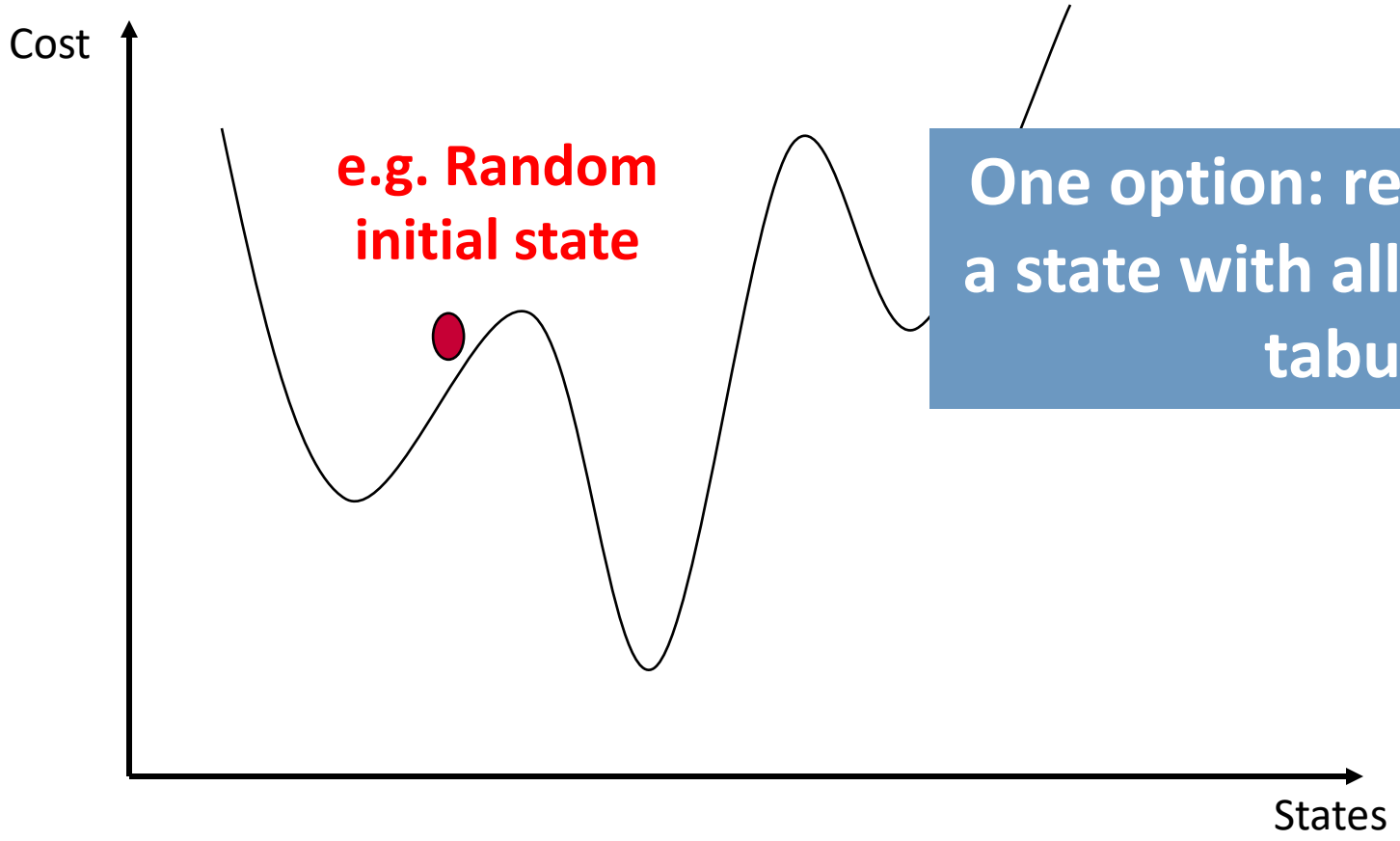


Tabu Search + Greedy hill climbing (minimization)



Want to escape while still using greedy selection

Will not work for every random initial state



One option: restart if end in a state with all neighbors on tabu list

Tabu Search

Properties:

- As the size of the Tabu list grows, hill-climbing will asymptotically become “non-redundant” (won’t look at the same state twice)

- In practice, a reasonable sized tabu list (say 100 or so) improves the performance of hill climbing in many problems

Tabu Search

- Proposed independently by Fred Glover (1986) and Pierre Hansen (1986)
  - “A meta-heuristic superimposed on another heuristic. The overall approach is to avoid entrapment in cycles by forbidding or penalizing moves which take the solution, in the next iteration, to points in the solution space previously visited (hence Tabu)”
  - Good overview article of Glover, Laguna and Marti (albeit from 2007): https://www.uv.es/~rmarti/paper/docs/ts1.pdf

Fred Glover

Pierre Hansen

Tabu Search

Tabu Search (TS):

determine initial candidate solution s
While *termination criterion* is not satisfied:
  determine set N' of non-tabu neighbours of s
  choose a best improving candidate solution s' in N'
  update tabu attributes based on s'
  $s := s'$

Tabu search prevents returning quickly to same state.

- To implement: Keep fixed length queue (tabu list).
- Add most recent state to queue; drop oldest state.
- Never go to state that's on current tabu list.

Tabu Search

- Saves information according to the exploration process
- It will be used to limit the moves through the neighbourhood
- Structure of the neighbourhood of the solution varies from iteration to iteration
- Infeasible / worsening solutions can be accepted and evaluated to escape local minimum
- To prevent from cycling, recent moves are forbidden
- A Tabu list records forbidden moves, which are referred to as Tabu moves
- Allows exploitation of good solutions and exploration of unvisited region of the search space

Uses memory during search

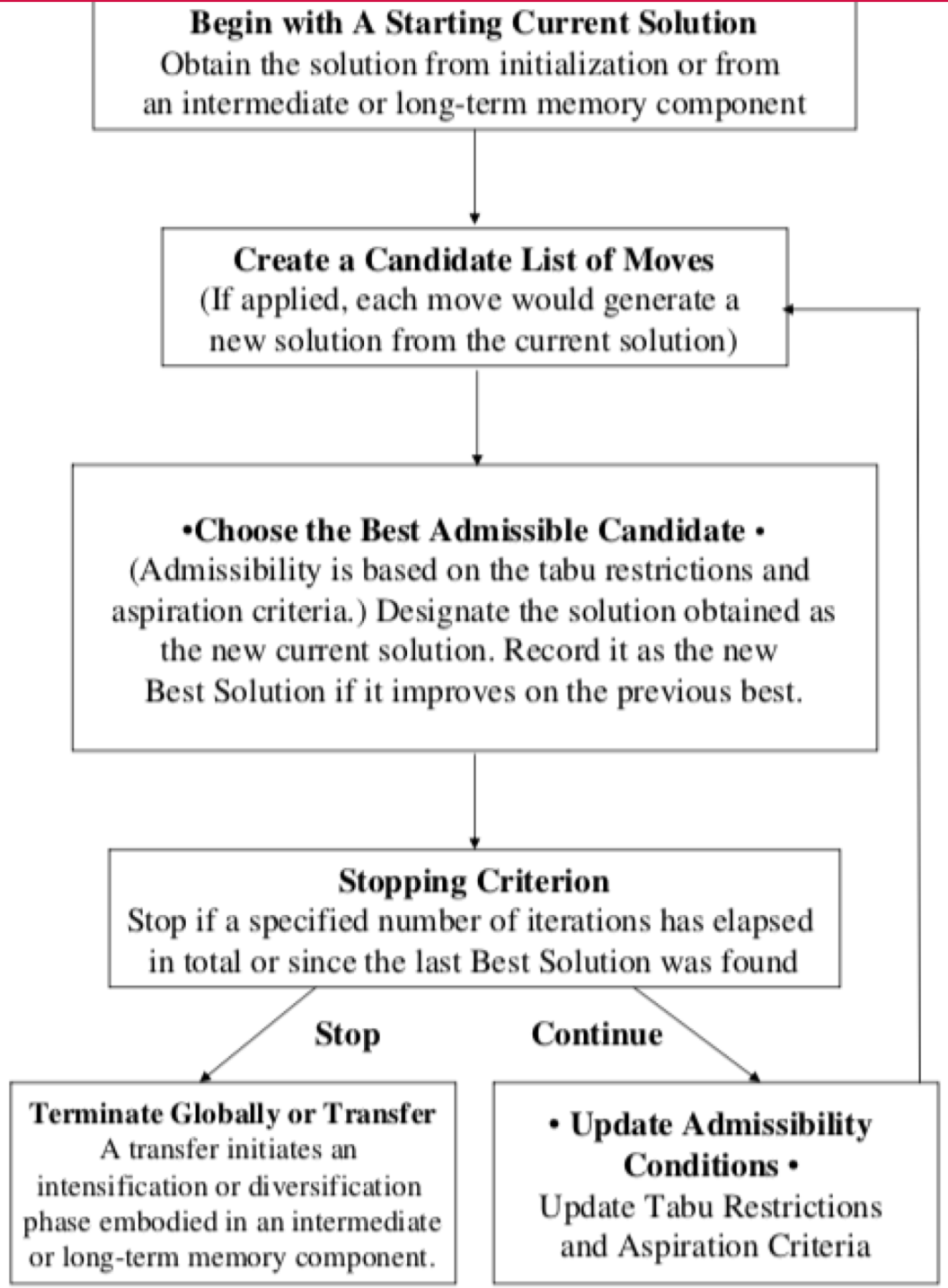
- Tabu move:
  - Not allowed to re-visit the exact same state that we've been before
  - Not allowed to return to the state that the search has just come from

- Memory issues:
  - Possibly too expensive to store every state visited (and to have to check through this list for each possible neighbor in each iteration)
  - Possibly too expensive to store the full state each time

Uses memory during search

- Solution:
  - Store list (queue) of fixed length *(tenure)*
    - Variants where length changes dynamically based on direction (towards improving solution or away from solution)
  - Store abstraction of state, or transitions/operations that changed the state
- Examples of abstraction:
  - TSP: Tabu the edges added in the last move → Search is forced to consider other edges for deletion
  - SAT: Store variables flipped
    - Similar to?

Tabu Search



Glover, Fred. "Tabu search: A tutorial." *Interfaces* 20, no. 4 (1990): 74-94.

Tabu Search & SAT Example

Size of Tabu list = 3

- Start with a random candidate solution (Tabu list = [])
  - X1 = F, X2 = F, X3 = T, X4=F, X5 = T, X6 = F
- Select the best variable (not in the Tabu list)
  - Best var: X2 (Flip variable)
  - X1 = F, X2 = T, X3 = T, X4=F, X5 = T, X6 = F
  - Update Tabu List = [X2]
- Select the best variable (not in the Tabu list)
  - Best var: X4 (Flip variable)
  - X1 = F, X2 = T, X3 = T, X4=T, X5 = T, X6 = F
  - Update Tabu List = [X2, X4]

Tabu (or invalid) variables:
X2

Tabu Search & SAT Example

Size of Tabu list = 3

- Select the best variable (not in the Tabu list)
  - Best var: X1 (Flip variable)
  - X1 = T, X2 = T, X3 = T, X4=T, X5 = T, X6 = F
  - Update Tabu List = [X2, X4, X1]

Tabu (or invalid) variables:
X2, X4

- Select the best variable (not in the Tabu list)
  - Best var: X6 (Flip variable)
  - X1 = T, X2 = T, X3 = T, X4=T, X5 = T, X6 = T
  - Update Tabu List = [X4, X1, X6]

Tabu (or invalid) variables:
X2, X4, X1

- Select the best variable (not in the Tabu list)
  - Best var: X5 (Flip variable)
  - X1 = T, X2 = T, X3 = T, X4=T, X5 = F, X6 = T
  - Update Tabu List = [X1, X6, X5]

Tabu (or invalid) variables:
X4, X1, X6

Implementation

- Van Hentenryck discusses one option which is to store for each possible move, the earliest next iteration that that move will be allowed

- So if our ”list” is of length L, then the current move will be allowed again after iteration *it_curr + L*

  where *it_curr* is the current iteration number

- When computing our heuristic value for neighbors, we only consider those neighbors that are not tabu

- Note in this variant then we don’t get stuck, if no neighbors are valid we increase the iteration and try again, eventually the iteration number will be greater than the tabu value for a neighbor move

Memory in Tabu Search

- Tabu Search uses mainly two types of memory

- Short-term memory
  - Recent moves
  - Structure where Tabu moves are stored
  - Avoid cycling
  - Can also store the objective value before and after the move, if move resulted in worse objective value we may wish to allow it's reverse with a certain probability

- Long-term memory
  - Frequency-based: Number of iterations that "solution components" have been present in the current solution.
  - Can also include an ACO type memory where initial solutions are biased towards good nodes, or neighbour quality can take into account this information.

Diversification: choose nodes that haven't been explored less frequently

Intensification: choose nodes that have been "good"

Use of Memory in Tabu Search

- Use of memory leads to learning
- Prevent the search from repeating moves
- Explore the unvisited area of the solution space
- By using memory to avoid certain moves, Tabu Search can be seen as global optimizer rather than local

Important caveats to “non-repeating” moves for search

- Can’t guarantee non-repeating moves due to fixed length

- Will also **prevent** the search from exploring some ***new*** states for L iterations if storing the move as tabu rather than the entire state!
  - Won’t consider candidate solutions containing the tabu move
  - E.g. SAT Example from earlier:
    - Best var: X5 (Flip variable)
    - X1 = T, X2 = T, X3 = T, X4=T, X5 = T, X6 = T
    - Update Tabu List = [X1, X6, X5]
  - So won’t consider the following states in the next iteration even though haven’t visited them:

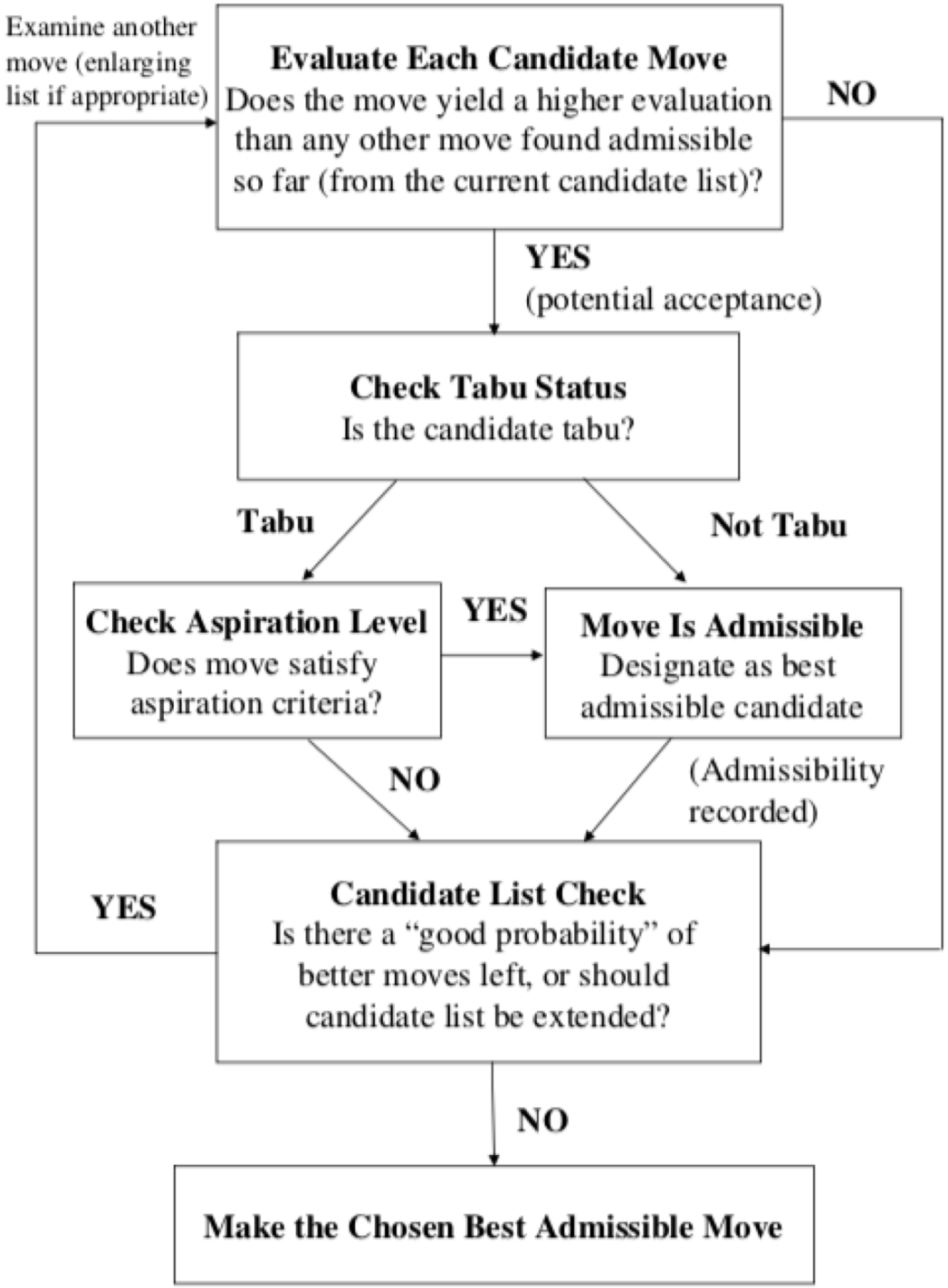
X1 = F, X2 = T, X3 = T, X4=T, X5 = T, X6 = T

X1 = T, X2 = T, X3 = T, X4=T, X5 = T, X6 = F

Aspiration

- If a move which is Tabu will produce a **better** solution than the **current best**, the Tabu status should be overruled
- This will be performed using ***aspiration*** level conditions
- **Aspiration criteria**: accepting an improving solution even if generated by a Tabu move
- A Tabu move becomes admissible if it yields a solution that is better than an aspiration
- A Tabu move becomes admissible if the direction of the search (improving or non-improving) does not change

Tabu Search



Glover, Fred. "Tabu search: A tutorial." *Interfaces* 20, no. 4 (1990): 74-94.

Basic Tabu Search Algorithm

Step 1: Choose an initial solution i in S. Set i*=i and k=0.

Step 2: Set k=k+1 and generate a subset V of solutions in N(i,k) such that the Tabu conditions are not violated or the aspiration conditions hold.

Step 3: Choose a best j in V and set i=j.

Step 4: If f(i) < f(i*) then set i* = i. // Update best (minimization)

Step 5: Update Tabu and aspiration conditions.

Step 6: If a stopping condition is met then stop. Else go to Step 2.

Stopping Conditions

- No feasible solution in the neighborhood of solution i
- The number of iterations since the last improvement of i* is larger than a specified number
- Evidence can be given that an optimum solution has likely been obtained