Topics in Local Search

Thomas Stützle & Irina Dumitrescu

{tom,irina}@intellektik.informatik.tu-darmstadt.de
http://www.intellektik.informatik.tu-darmstadt.de/~tom,~irina.

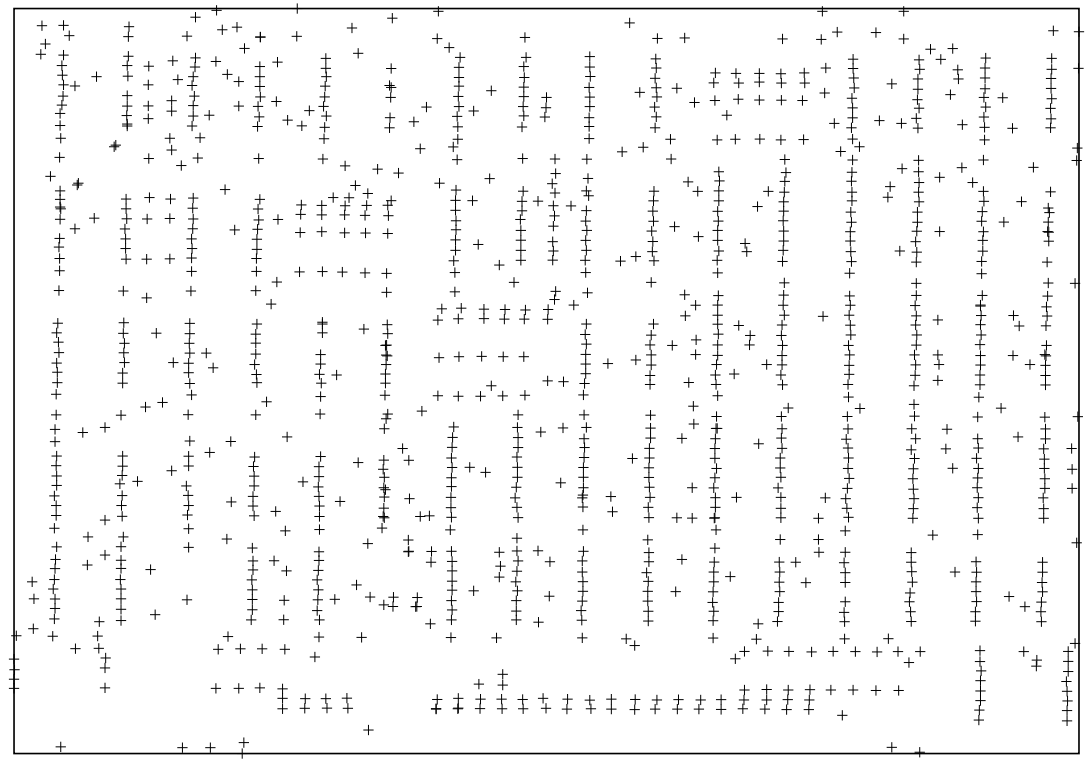
Darmstadt University of Technology
Department of Computer Science
Intellectics Group

Outline

- local search
- efficiency
  - cost function evaluation
  - neighborhood pruning
  - speed-up techniques / data structures
- large neighborhoods
  - variable depth search
  - very large neighborhood search
- concluding remarks

Example problem — TSP

- given: fully connected, weighted Graph $G = (V, E, d)$
- goal: find shortest Hamiltonian cycle
- hardness: $\mathcal{NP}$-hard
- interest: standard benchmark problem for algorithmic ideas



Local search

Ingredients

- (candidate) solution representation, search space definition $\mathcal{S}$

  TSP: set of all possible permutations of the city indices

- solution set $\mathcal{S}' \subseteq \mathcal{S}$

  TSP: set of all shortest Hamiltonian cycles (tours)

- cost function $f : S \mapsto \mathbb{R}_0^+$

  TSP: sum of the weights of the edges in a tour

- neighborhood relation $\mathcal{N} \subseteq \mathcal{S} \times \mathcal{S}$ TSP: e.g. k-exchange neighborhood;

  two tours differ in (at most) k edges

- an examination scheme of the neighborhood

  how to search neighborhood and to choose a new solution

Local search

Main issues

- neighborhood definition
  - problem specific
  - essential influence on efficiency and effectiveness of local search
  - tradeoff: size and solution quality vs. time to search
- neighborhood examination mechanism
  - in which order to search neighborhood
  - which neighboring solution becomes new one (pivoting rule)
- remark: neighborhoods are typically defined through moves that are applicable to solutions

Disclaimer

in this presentation we only talk about issues when implementing iterative improvement algorithms

procedure *Iterative improvement*
  while $\{s' \mid s' \in \mathcal{N}(s), f(s') < f(s)\} \neq \emptyset$ **do**
    $s \leftarrow s'$
  end

Advantages of local search

Why can local search be good (trivial reasons)?

- cost of generating neighboring solutions
  - typically, for generating a neighboring solution the computational complexity is much lower than generating a new solution from scratch
  - for evaluating a neighboring solution, it often does not need to generate it explicitly at all
- cost of evaluating neighboring solutions
  - typically Δ-evaluation can be done in a computational cost that is much less than computing solution cost from scratch

Disadvantages of local search?

- iterative improvement may take exponential time in the worst case
  but usually this occurs only rarely and for few problems
- exponential increase of the number of local minima with instance size
- short-sightedness of local search
- general: problem of local optimality

Experience has shown that the disadvantages are for many problems by far outweighted by the advantages

Pivoting rules

- gives a rule which of the neighboring solutions is accepted
  - best improvement
  - first improvement
  - (worst improvement) *please, don't use this one*
- "checkout-time"
- it is problem dependent, which pivoting rule results in better quality solutions or gives place to faster local search algorithms
- pivoting rules can have significant influence on the performance of local search algorithms

Example where first is faster than best

- TSP, `2-opt`, averages over 10 local searches (CPU:UltraSparc 300MHz)

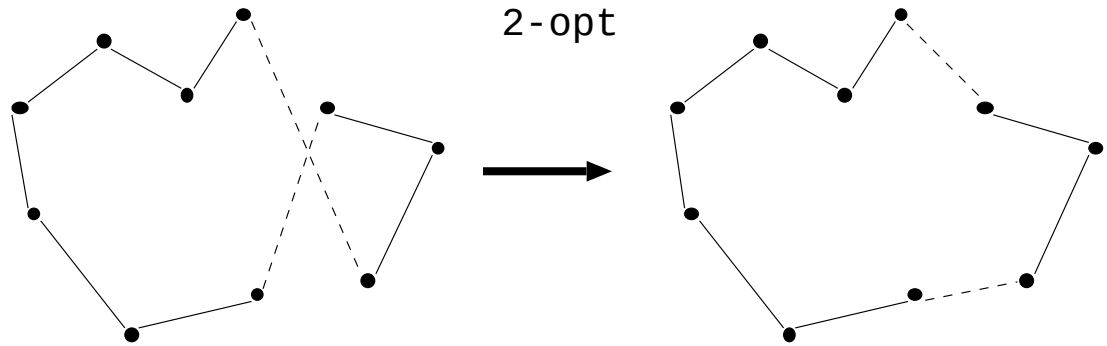
	random starting solution					
	best imp			first imp		
	secs	No.moves	Δ_{avg}	secs	No.moves	Δ_{avg}
`d198`	0.93	220	4.8%	0.038	390	5.2%
`lin318`	4.35	380	7.9%	0.12	680	14.3%
`pcb442`	11.60	500	10.9%	0.23	950	11.6%
`rat783`	72.06	750	10.0%	0.84	1820	11.2%
`pcb1173`	—	—	—	1.99	2730	13.3%
`fl1577`	—	—	—	4.74	2900	12.1%
`pr2392`	—	—	—	10.1	5790	13.9%

Neighborhood examination

- order of searching the neighborhood
  - deterministic order
  - random order
- where to continue the local search after an exchange
  - continue from where you are
  - restart from where you started scanning the neighborhood

$\rightsquigarrow$ best variant needs to be determined in an experimental way

Δ-evaluation — TSP



- compute cost of a `2-opt` move in constant time ($\mathcal{O}(1)$) as

$$\Delta_{ij} = d(\{i,j\}) + d(\{s(i),s(j)\}) - d(\{i,s(i)\}) - d(\{j,s(j)\})$$

  i, s(i): city i and its successor in the tour
  j, s(j): city j and its successor in the tour

- cost of evaluation function evaluation from scratch: linear time ($\mathcal{O}(n)$)

Example: QAP

- given: n objects and n locations with
  - a_{ij} : flow from object i to object j
  - $d_r{}^s$: distance between location r and location s
- goal: find an assignment (i.e. a permutation) of the n objects to the n locations that minimizes

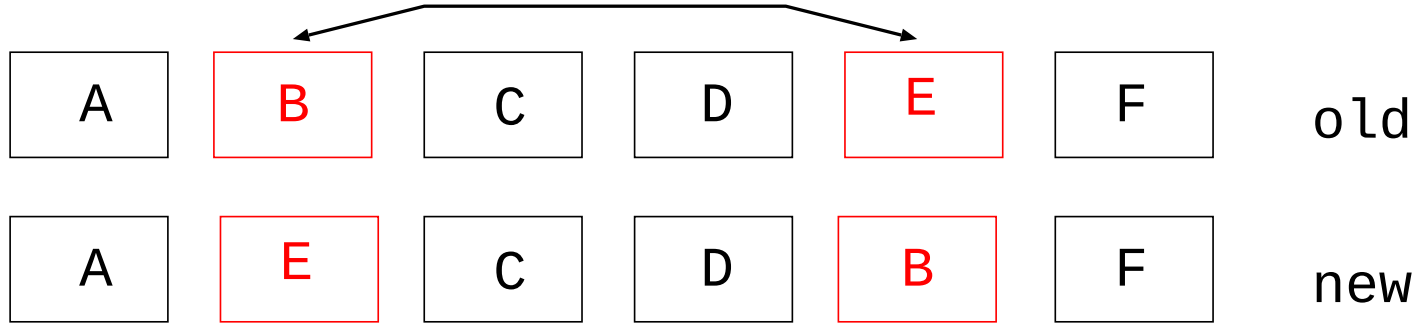
$$\min_{\pi \in \Pi(n)} \sum_{i=1}^{n} \sum_{j=1}^{n} a_{ij} d_{\pi(i)\pi(j)}$$

  $\pi(i)$ gives location of object i
- interest: it is among the "hardest" combinatorial optimization problems; several applications

Example: QAP

basic neighborhood for the QAP

- LocalSearch: `2-opt`



- computation of the cost function from scratch in $\mathcal{O}(n^2)$
- for `2-opt` we can again use Δ-evaluation
  $\rightsquigarrow$ a neighboring solution can be evaluated in $\mathcal{O}(n)$
- one full neighborhood scan can be completed in $\mathcal{O}(n^3)$

Exchange of objects r, s

for asymmetric instances

$$\Delta(\pi, r, s) = a_{rr} \cdot (d_{\pi_s \pi_s} - d_{\pi_r \pi_r}) + a_{rs} \cdot (d_{\pi_s \pi_r} - d_{\pi_r \pi_s}) + a_{sr} \cdot (d_{\pi_r \pi_s} - d_{\pi_s \pi_r}) + a_{ss} \cdot (d_{\pi_r \pi_r} - d_{\pi_s \pi_s}) + \sum_{k=1, k \neq r,s}^{n} (a_{kr} \cdot (d_{\pi_k \pi_s} - d_{\pi_k \pi_r}) + a_{ks} \cdot (d_{\pi_k \pi_r} - d_{\pi_k \pi_s}) + a_{rk} \cdot (d_{\pi_s \pi_k} - d_{\pi_r \pi_k}) + a_{sk} \cdot (d_{\pi_r \pi_k} - d_{\pi_s \pi_k}))$$

for symmetric instances

$$(1)\ \Delta(\pi, r, s) = 2 \cdot \sum_{k=1, k \neq r,s}^{n} (a_{sk} - a_{rk}) \cdot (d_{\pi(r)\pi(k)} - d_{\pi(s)\pi(k)})$$

Fast update

- π' results from π by exchanging objects r, s
- computation of $\Delta(\pi', u, v)$, with $\{u, v\} \cap \{r, s\} = \emptyset$

$$\Delta(\pi', u, v) = \Delta(\pi, u, v) + (a_{ru} - a_{rv} + a_{sv} - a_{su}) \cdot (d_{\pi_s \pi_u} - d_{\pi_s \pi_v} + d_{\pi_r \pi_v} - d_{\pi_r \pi_u}) + (d_{\pi_u \pi_s} - d_{\pi_v \pi_s} + d_{\pi_v \pi_r} - d_{\pi_u \pi_r}) \cdot (a_{ur} - a_{vr} + a_{vs} - a_{us})$$

Fast update

- fast update can be used within best improvement local search (ie. also tabu search)
- requires: additional memorization of the $\Delta(\pi, i, j)$ value for all pairs r, s in a table
- first local search iteration in $\mathcal{O}(n^3)$ for initializing the table
- in the subsequent iterations exchanges can be computed in $\mathcal{O}(1)$
- exception: objects that were moved in previous iteration

Example — 2-opt for QAP

- local search variants of 2-opt, average results over 100 restarts; times measured on a Pentium III 500MHz

	best Imp.			first Imp			first Imp+dlbs		
	secs	moves	Δ_{avg}	secs	moves	Δ_{avg}	secs	moves	Δ_{avg}
`tai50b`	0.06	56	6.9%	0.08	189	7.1%	0.04	186	7.2%
`tai60b`	0.12	72	8.3%	0.16	266	7.4%	0.08	261	7.5%
`tai80b`	0.33	100	6.3%	0.43	368	5.9%	0.22	356	6.0%
`sko72`	0.21	77	2.6%	0.27	210	2.5%	0.13	202	2.7%
`sko80`	0.31	89	2.3%	0.42	249	2.1%	0.19	240	2.3%
`sko90`	0.46	105	2.2%	0.62	288	2.2%	0.28	276	2.4%

don’t look bits

- a technique that allows to focus the local search around the part where potentially there can happen something
- allows to reduce the checkout time
- only applicable with first-improvement pivoting rule
- proceeds as
  - associate to each "component" a don’t look bit
  - if don’t look bit is zero, the component can be used in an outer loop of an improvement search
  - if no improving move is found for the component: set its don’t look bit to one
  - if a component is involved in a move: set don’t look bit to zero
  - if for the "outer-loop component" no improving move is found: set its don’t look bit to one

don't look bits

procedure iterative improvement

  for $i = 1$ **to** n **do**

    if $\texttt{dlb}[i] = 1$ **then** continue

    improve_flag $\leftarrow$ *false*

    for $j = 1$ **to** n **do**

      CheckMove(i, j)

      if move improves **then**

        ApplyMove(i, j); $\texttt{dlb}[i] \leftarrow 0$, $\texttt{dlb}[j] \leftarrow 0$

        improve_flag $\leftarrow$ *true*

    endfor

    if *improve_flag* $=$ *false* **then** $\texttt{dlb}[i] \leftarrow 1$

  end

end iterative improvement

don’t look bits

- often: significant speed-up at only low loss of solution quality
- integration possibilities between perturbation and local search in ILS
  - reset don’t look bits to zero only of "moved" solution components in a perturbation
- same possibility is available for memetic algorithms after applying recombination or mutation
- some SLS methods do not allow for an easy integration of don’t look bits

Perturbation — Speed, ILS for TSP

- compare No. local searches (here, 3-opt) in fixed computation time
- $\#LS_{RR}$: No. local searches with random restart
- $\#LS_{1\text{-}DB}$: No. local searches with one double bridge move as Perturbation
- $\#LS_{1\text{-}DB}/\#LS_{RR}$: factor between $\#LS_{1\text{-}DB}$ and $\#LS_{RR}$
- time limit: 120 sec on a Pentium II 266 MHz PC

instance	$\#LS_{RR}$	$\#LS_{1\text{-}DB}$	$\#LS_{1\text{-}DB}/\#LS_{RR}$
kroA100	17507	56186	3.21
d198	7715	36849	4.78
lin318	4271	25540	5.98
pcb442	4394	40509	9.22
rat783	1340	21937	16.38
pr1002	910	17894	19.67
d1291	835	23842	28.56
fl1577	742	22438	30.24
pr2392	216	15324	70.94
pcb3038	121	13323	110.1
fl3795	134	14478	108.0
rl5915	34	8820	259.4

Neighborhood pruning

Example: TSP, 2-opt local search

- important property: for any improving `2-opt` move, there is at least one node that is incident to an edge e that is replaced by a different edge e' with lower weight
- fixed radius nearest neighbour search
  - consider both tour neighbours of a node v_i, say v_j
  - search around v_i for nodes v_k for which holds $d(v_i, v_k) < d(v_i, v_j)$
  - for each such city v_k delete unique edge to make feasible `2-opt` move and test for improvement
  - if fixed radius near neighbor searches for all nodes are unsuccessful, the tour is `2-opt`

Neighborhood pruning

- support fixed radius search by appropriate data structures
- nearest neighbor lists for each city
  - for each node v_i $nl(v_i, r)$ gives the r–nearest neighbour of v_i
  - several possibilities available of how to construct candidate sets

neighborhood pruning applicable similarly to many geometric problems; for other problems often more complicated than for TSP or not possible at all

Example results: TSP

- timings for 1000 local searches with `2-opt` and `3-opt` variants from random initial solutions on a Pentium III 500 MHz CPU. **std**: no speed-up techniques; **fr+cl**: fixed radius and unbounded candidate lists, **dlb**: don't look bits

	`2-opt`-**std**		`2-opt`-**fr+cl**		`2-opt`-**fr+cl+dlb**		`3-opt`-**fr+cl+dlb**	
instance	Δ_{avg}	t_{avg}	Δ_{avg}	t_{avg}	Δ_{avg}	t_{avg}	Δ_{avg}	t_{avg}
`kroA100`	8.9	1.6	6.4	0.5	6.6	0.4	2.4	4.3
`d198`	5.7	6.4	4.2	1.2	4.3	0.8	1.4	30.1
`lin318`	10.6	22.1	7.5	2.1	7.9	1.5	3.4	65.5
`pcb442`	12.7	55.7	7.1	2.9	7.6	2.2	3.8	63.4
`rat783`	13.0	239.7	7.5	7.5	8.0	5.8	4.2	213.8
`pr1002`	12.8	419.5	8.4	13.2	9.2	9.7	4.6	357.6
`pcb1173`	14.5	603.1	8.5	16.7	9.3	12.4	5.2	372.3
`d1291`	16.8	770.3	10.1	16.9	11.1	12.4	5.5	377.6
`fl1577`	13.6	1251.1	7.9	25.8	9.0	19.2	4.0	506.8
`pr2392`	15.0	2962.8	8.8	65.5	10.1	49.1	5.3	878.1

Efficient computation of move values

Example: SAT, GSAT local search architecture

- 1-opt neighbourhood
- one of the first local search algorithms for SAT
- best improvement pivoting rule

Local Search for SAT

procedure *local search for SAT*
  input *CNF formula* Φ, *maxTries, maxSteps*
  output *model for* Φ *or "no solution found"*
  for i := 1 **to** *maxTries* **do**
    s := *initAssign*(Φ);
    for j := 1 **to** *maxSteps* **do**
      if s satisfies Φ **then return** s;
      else
        x := *chooseVariable*(Φ, s);
        s := s with truth value flipped for x;
      end if
    end for
  end for
  return "no solution found";
end *local search for SAT*

Implementation issues

- for each iteration one needs to compute scores of variables
- simple approach
  - recompute scores after each iteration from scratch
  - requires effort in $\mathcal{O}(m \cdot CL(n))$

    m: number of clauses, $CL(n)$: bound on maximum clause length
- efficient computation of flip effects required
- idea
  - use dynamic update of the scores
  - use appropriate data structures to allow for the dynamic update

Implementation issues

- central observation
  - only the score of variables x' is affected by flipping a variable x that occur in a same clause as variable x
  - for an update of the score only clauses are interesting in which the flipped variable x occurs

$$C_{dep}(\Phi, x) = \{c \text{ is a clause of } \Phi \mid x \text{ appears in clause } c$$

- data structure
  - each variables has a list of clauses where it occurs
  - for each variable store truth value, score; for each clause store it satisfaction status

Score update

- the score of variable x changes its sign
- go through all the clauses $c \in C_{dep}(\Phi, x)$ and for all variables in each clause do
  - if the clause has become unsatisfied by flipping x, then increase the score of the other variables by one
  - if the clause was unsatisfied and has become satisfied, then decrease score of all other variables by one
  - if flipping x makes two variables instead of one satisfying the clause, then search the other one and increase its score by one

Analogous update schemes are essential for many problems like graph coloring, time tabling, set covering, etc.

Large neighborhoods

- several local search algorithms search large, typically exponentially sized neighborhoods
- exploration of the neighborhoods through appropriate techniques typically possible in polynomial time either through
  - insight into neighborhood structure and searching it with exact algorithms
  - a heuristically guided search in the neighborhood
- advantages: typically much better solution quality reachable

Variable depth search algorithms

- complex moves are build as being a concatenation of a number of simple moves
- the number of simple moves composing a complex one is variable and determined based on gain criteria
- the simple moves need not be independent of each other
- termination is guaranteed through additional conditions on the simple moves

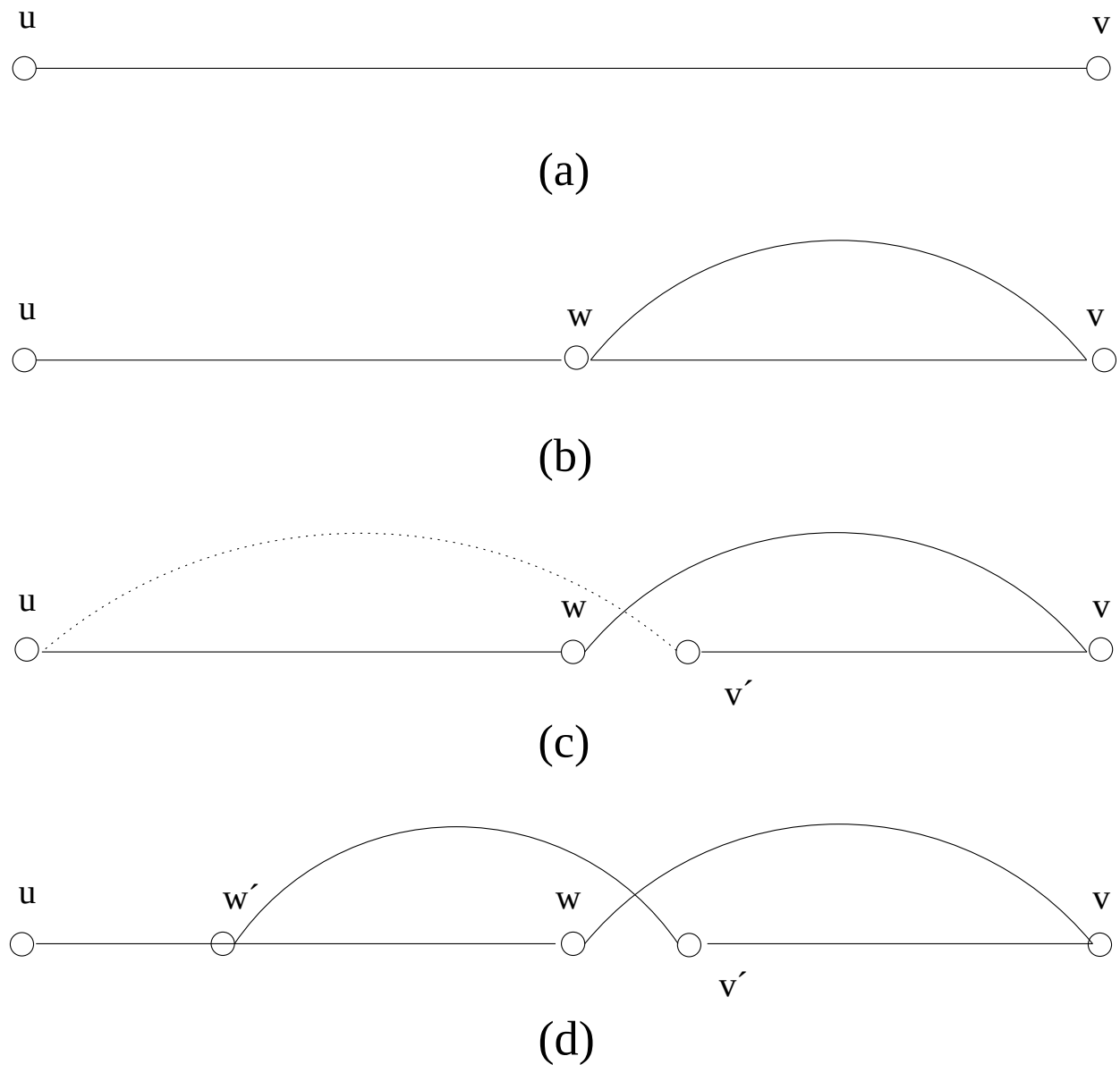
example

- Lin-Kernighan algorithm for TSPs

Lin-Kernighan for TSPs

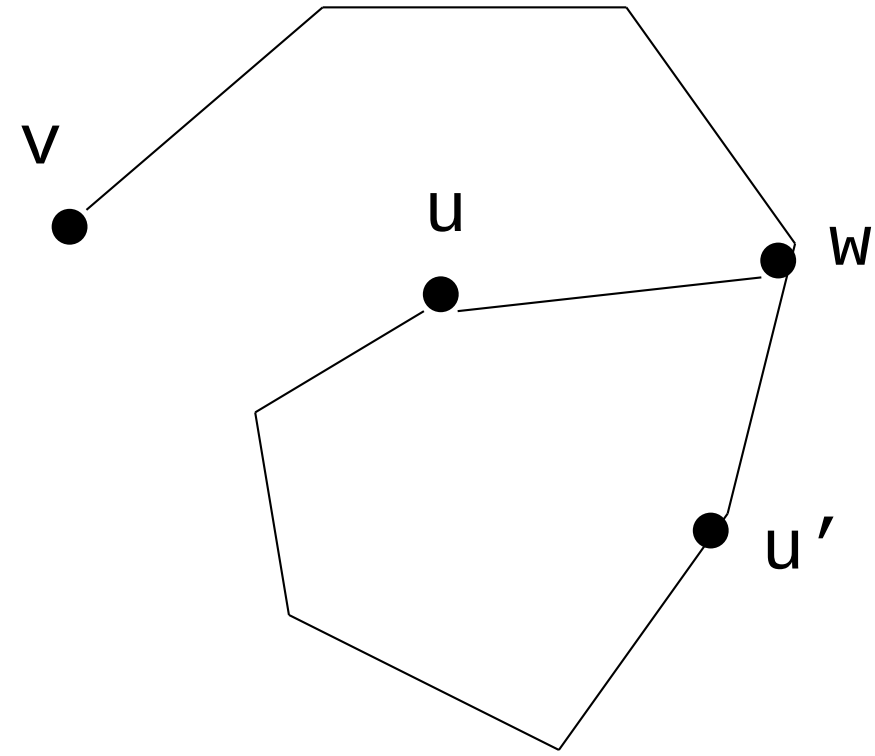
- at each complex search step a set of edges $X = \{x_1, \ldots, x_r\}$ is deleted and another set $Y = \{y_1, \ldots, y_r\}$ is added to a tour
- the number of edges r is determined dynamically
- the two sets X and Y are constructed iteratively, element by element
- edges x_i and y_i as well as y_i and x_{i+1} must share an endpoint (sequential moves)
- at any point in the search there needs to be an edge y_i' such that the complex step defined by $X = \{x_1, \ldots, x_i\}$ and $Y = \{y_1, \ldots, y_i'\}$ is a feasible tour
- illustration: δ-path

Lin-Kernighan for TSPs



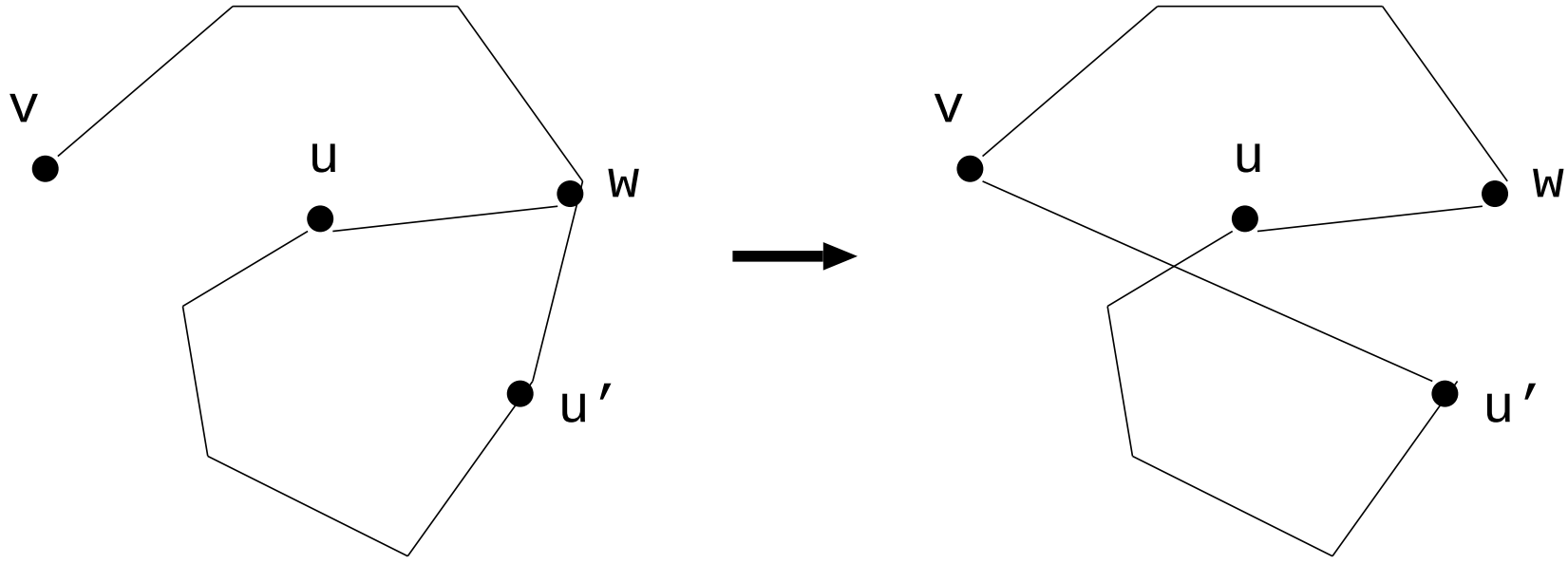
δ-path

- δ-path: (spanning tree plus one edge)

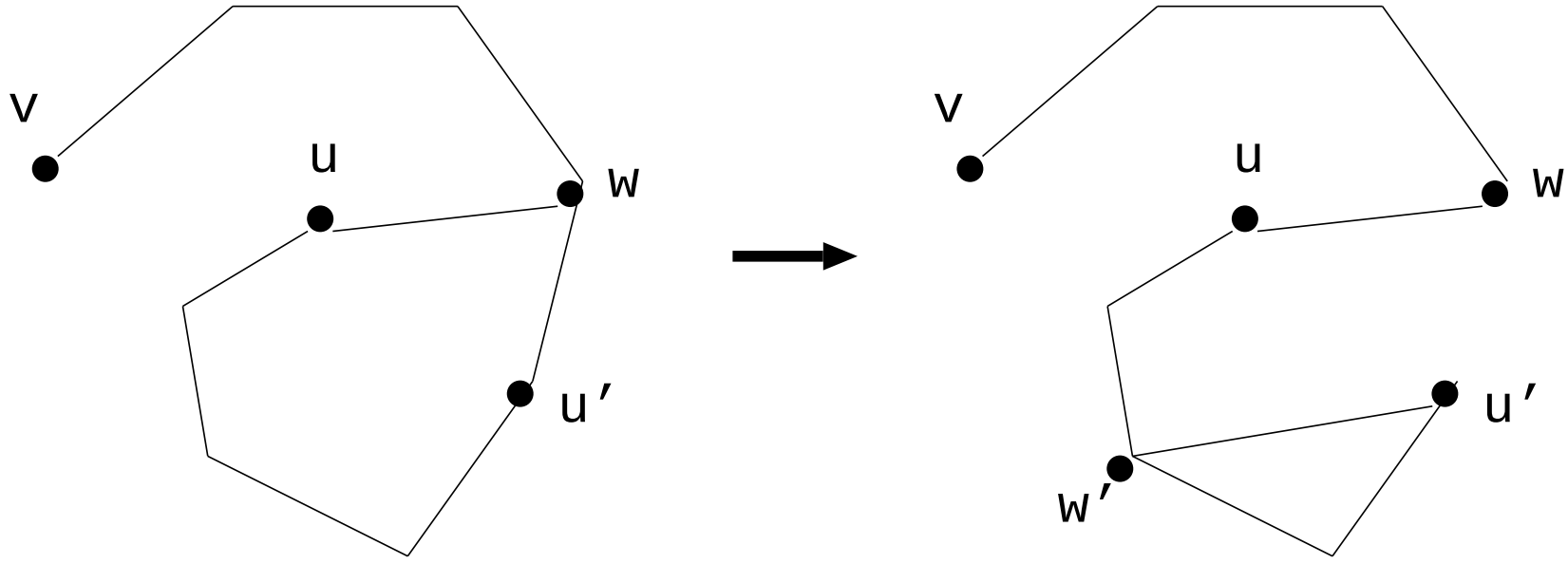


a delta-path

- convert a δ-path into a tour



- new δ-path from previous δ-path



Limitations on moves

- length restrictions
  - edges that are included in set Y (added edges) may not be deleted anymore
  - edges that are included in set X (deleted edges) may not be added again

  $\rightsquigarrow$ bounds the depth of the search to a maximum of n moves
- cost restrictions
  - stop the construction process of the complex move if the resulting δ-path has higher weight than the shortest tour found in the process

Search guidance in LK

- at each step, try to include a least costly possible edge y_i
- if no improved complex move is found
  - apply backtracking on the first and second level of the construction steps (choices of x_1, y_1, x_2, y_2)
  - consider alternative choices in order of increasing weight of candidates up to a maximum number of candidates
  - at the last level, consider different starting nodes for search
  - backtracking assures final tour to be `2-opt` and `3-opt`
- important are techniques for pruning the search

Remarks

- Lin-Kernighan algorithm is best performing local search for TSP
- many variants of the algorithm are available (see also recent DIMACS challenge)
- an efficient implementation requires sophisticated data structures (several articles available on this subject)
- implementation is quite time consuming, but
- at least three very good implementations are publically available (concorde, Helsgaun, Neto)

variable depth search algorithms are now available for many problems and for many they show excellent performance

Ejection chains

- similar approach as in variable depth search algorithms
- differences concern mainly that ejection chains allow for more flexibility in move generation
- complex moves are composed of a sequence of dependent, simple moves
- “Ejection moves”: moves that allow to do a transition to a different solution by ejecting some solution components
- “trial moves”: moves that try to restore feasible solutions

Conclusions (1)

- implementation aspects
  - efficient evaluation of cost functions (Δ–evaluation etc.)
  - seemingly minor implementation details can have significant influence on search performance (pivoting rules, order in which neighborhoods are scanned etc.)
  - exploitation of problem specific properties can improve strongly the speed of local search
  - appropriate data structures are essential for efficient implementations of local search algorithms

But: they do not make up for a poor choice of a neighborhood

Conclusions (2)

- complex (large-scale) neighborhoods
  - allow to obtain better solution quality in a single local search step than simple exchange neighborhoods
  - often explore exponentially sized neighborhoods but that are explored typically in polynomial time
  - often relatively complex to implement them efficiently
  - often require deep knowledge about the problem for their development
  - but for several problems they are the (by far) best performing local search algorithms

Literature

• E.H.L. Aarts and J.K. Lenstra, editors. *Local Search in Combinatorial Optimization*. John Wiley & Sons, 1997.

• J.L. Bentley. Fast Algorithms for Geometric Traveling Salesman Problems. *ORSA Journal on Computing*, 4(4):387–411, 1992.

• R. K. Congram and C. N. Potts and S. L. Van de Velde. An Iterated Dynasearch Algorithm for the Single–machine Total Weighted Tardiness Scheduling Problem. Working paper, *Faculty of Mathematical Studies, University of Southampton*, 1998.

• F. Glover. Ejection Chains, Reference Structures and Alternating Path Methods for Traveling Salesman Problems. *Discrete Applied Mathematics*, 65:223–253, 1996.

• D.S. Johnson and L.A. McGeoch. The Travelling Salesman Problem: A Case Study in Local Optimization. In E.H.L. Aarts and J.K. Lenstra, editors, *Local Search in Combinatorial Optimization*, pages 215–310. John Wiley & Sons, 1997.

• B.W. Kernighan and S. Lin. An Efficient Heuristic Procedure for Partitioning Graphs. *Bell Systems Technology Journal*, 49:213–219, 1970.

• S. Lin and B.W. Kernighan. An Effective Heuristic Algorithm for the Travelling Salesman Problem. *Operations Research*, 21:498–516, 1973.

• B. Selman, H. Levesque, and D. Mitchell. A New Method for Solving Hard Satisfiability Problems. In *Proceedings of the 10th National Conference on Artificial Intelligence*, pages 440–446. MIT press, 1992.

• É.D. Taillard. Comparison of Iterative Searches for the Quadratic Assignment Problem. *Location*