ΟΙΚΟΝΟΜΙΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ATHENS UNIVERSITY OF ECONOMICS AND BUSINESS

Special Topics on Algorithms

Local Search

Vangelis Markakis

markakis@gmail.com

Outline

- Introduction to Local Search Algorithms
  - Illustration for Vertex Cover
  - Illustration for TSP
- Tabu search
- Application to the MAX CUT problem

Local Search Algorithms

- Iterative algorithms that move from a current solution to a "nearby" one
- Starting from an initial feasible solution, they keep moving to new solutions by applying local structural changes
- They stop at a local optimum

Neighborhoods:

- To define a local search algorithm, we need to define the set of allowed moves
- Given a feasible solution S, let N(S) = set of neighboring solutions
- i.e., N(S) = the solutions we are allowed to examine, if we are at solution S

Local Search Algorithms

General scheme:

- Start from some feasible solution S_0
  - $t = 0$, $S_t = S_0$
- Repeat:
  - Select a new solution $S' \in N(S_t)$
  - $t{+}{+}$, $S_t = S'$
- Until the stopping criterion is met

Local Search Algorithms

- How do we select a neighbor of the current solution?
  - Need to define a selection criterion to pick a member of $N(S_t)$
  - Usual criterion: Best-pick (Pick the best neighbor according to the objective function)
    - This means we may need to explore all the neighboring solutions in order to identify the best
- What is the stopping criterion?
  - Suppose we have a minimization problem, and let Z be the objective function
  - If at iteration t, $Z(S_t) \leq Z(S)$ for every $S \in N(S_t)$, then S_t is a local optimum and we can terminate the algorithm there
  - In some cases, we may keep going, hoping to find an even better local optimum later

Local Search Algorithms

Classic best-pick version of local search (for minimization problems):

- Start from some feasible solution S_0
  - $t = 0$, $S_t = S_0$
- Repeat:
  - Select the best neighboring solution $S' \in N(S_t)$
  - If $Z(S') < Z(S_t)$
    - t++
    - $S_t = S'$
  - Else terminate and return S_t

Local Search Algorithms

Trade-offs in designing local search methods

- How shall we define the neighborhood of a given solution?
  - Most important decision to make
- We need N(S) to have relatively large cardinality
  - This increases the chances that we identify good quality solutions
- But we also do not want N(S) to be too large to keep the method fast
  - With best-pick, if N(S) is too large, we will spend too much time within every iteration
- Often one needs to identify by trial and error the best compromise

Performance of Local Search Algorithms

How good is a local optimum?

- Local optima may differ significantly from the global optimum
- In general, no guarantee that local optima are provably close to the global optimum
- But in many cases, we observe very good approximations in practice
- In few cases, we can also prove that local search achieves a good approximation guarantee

Local Search Algorithms

A simple illustration on (unweighted) Vertex Cover

- Need to define a notion of "neighborhood"

Neighbor relation: Given a solution S

- N(S) = neighboring vertex cover solutions
- $S' \in N(S)$ if it can be obtained from S by adding or deleting a single node

Algorithm:

- Given a graph G = (V, E), start with S = V (the full set of nodes)
- If there is a neighbor S' that is a vertex cover and has lower cardinality, replace S with S'

Local Search Algorithms

Cost of each iteration:

- Suppose the graph has n vertices and m edges
- Then $|N(S)| \leq n$ for every feasible solution S
  - You either delete a vertex in S, or add a vertex not in S
- Checking if a neighboring solution is a vertex cover needs O(m) time
- Total cost within each iteration = O(nm)

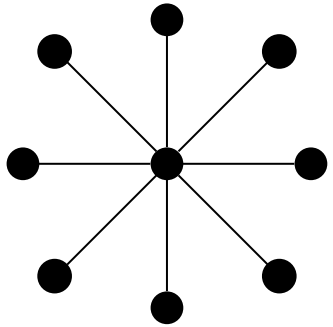
Number of iterations:

- For unweighted Vertex Cover, there are n possible values for the optimal vertex cover
- In each iteration, we reduce the current solution by 1 vertex
- Hence O(n) iterations

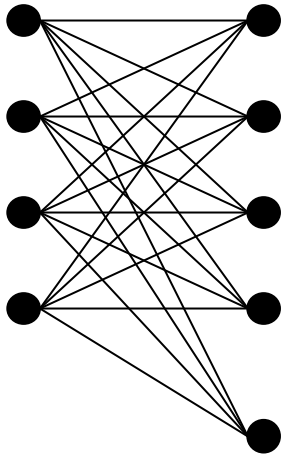
Corollary: The algorithm runs in total time $O(n^2m)$

Local optima in Vertex Cover

- How bad is the performance of local optima?



optimum = center node only
local optimum = all other nodes



optimum = all nodes on left side
local optimum = all nodes on right side

Local Search Algorithms

Local search for TSP

- Many ways to think about local moves
- Various local search heuristics tested on TSP over the years

1-1 exchanges:

- Each feasible solution corresponds to a cycle showing how we visit the cities
  - E.g., with S = (c_1, c_2, …, c_n, c_1), we mean that we first go to city c_1, then to c_2, …, and eventually back to c_1
- N(S) = all solutions produced by swaps between 2 cities
- Exchange between city i and city j:
  - (c_1, c_2, …, c_i, …, c_j, …, c_n, c_1) $\Rightarrow$ (c_1, c_2, …, c_j, …, c_i, …, c_n, c_1)

Local Search Algorithms

Local search for TSP

- Many ways to think about local moves
- Various local search heuristics tested on TSP over the years

Relocations:

- A different type of structural changes
- Pick a city and shift it from its position in the current ordering
- E.g., pick the city in position 3, and move it to the 7^{th} position in the ordering
- N(S) = all solutions produced by relocating 1 city

Local Search Algorithms

- In some cases, we can use multiple local search operators within an iteration
- For TSP, we could use both relocations and 1-1 exchanges

Pseudocode within each iteration:

- Find the best 1-1 exchange move S'
- If S' is better than the current solution, move to S'
- Otherwise, find the best relocation move S''
- If S'' is better than current solution, move to S''
- Otherwise, current solution is a local optimum

OR

- Find the best among: the best 1-1 exchange move and the best relocation move
- Move there if it improves the current solution

Local Search Algorithms

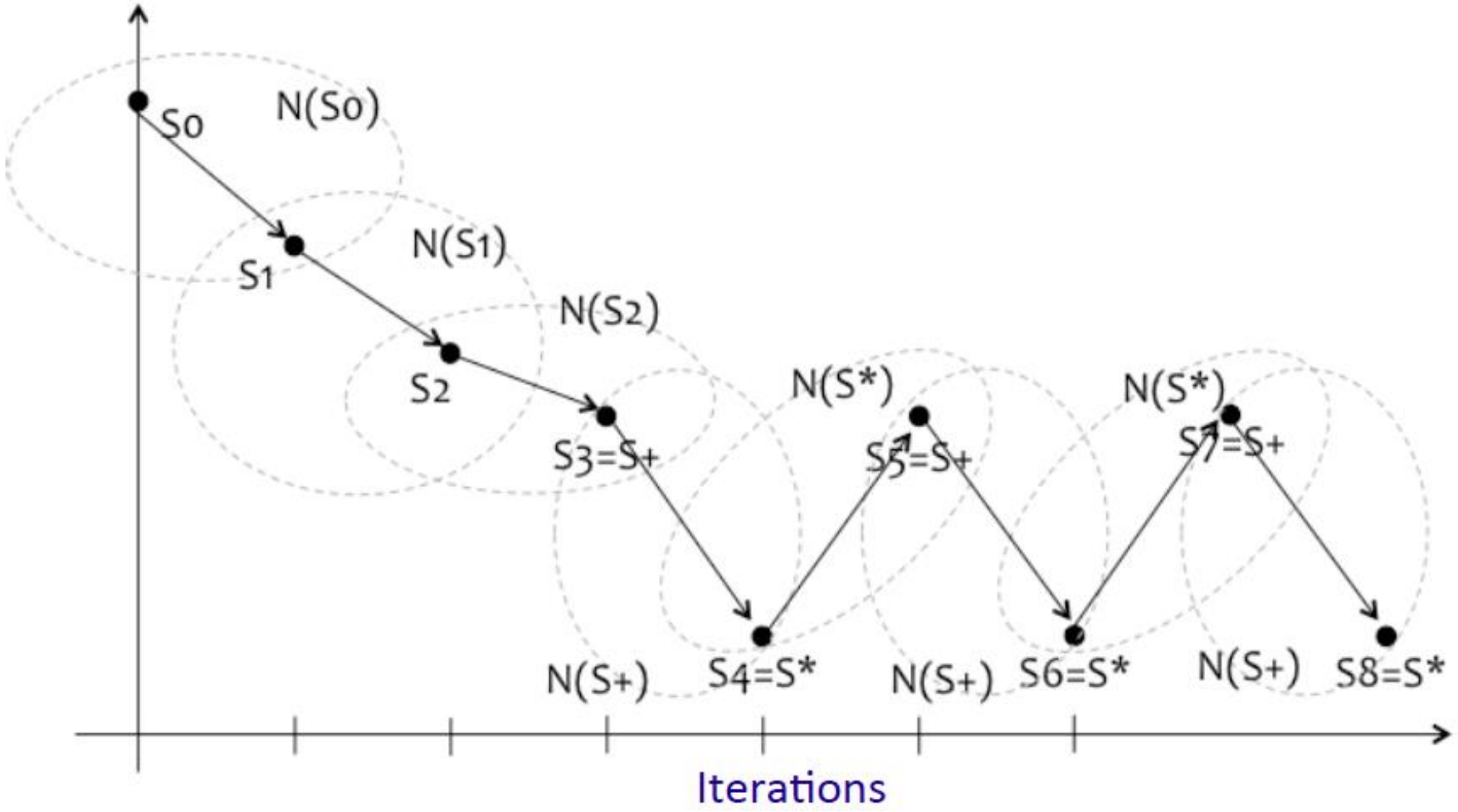
Yet more variants of local search

- For the versions we have seen so far, we stop the very first time we hit a local optimum
- We can keep moving to the best neighboring solution even if in some steps we move to a worse solution than the current one
- This way we may discover better local optima in the future

Danger:

- Suppose we find a local optimum S*
- In the next iteration, we move to a worse solution which increases the objective function, say $S_+ \in N(S^*)$
- From then on, we may have an oscillation between S_+ and S*
- We call this phenomenon cycling

Cycling in Local Search



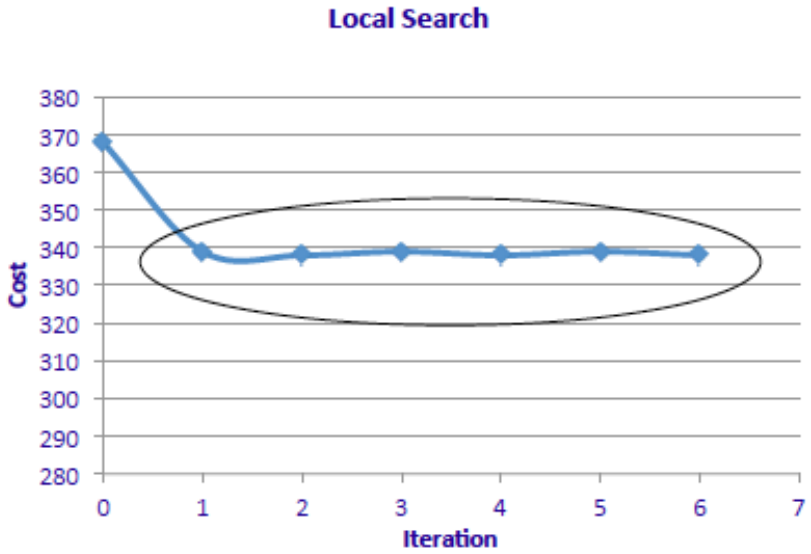
Tabu Search

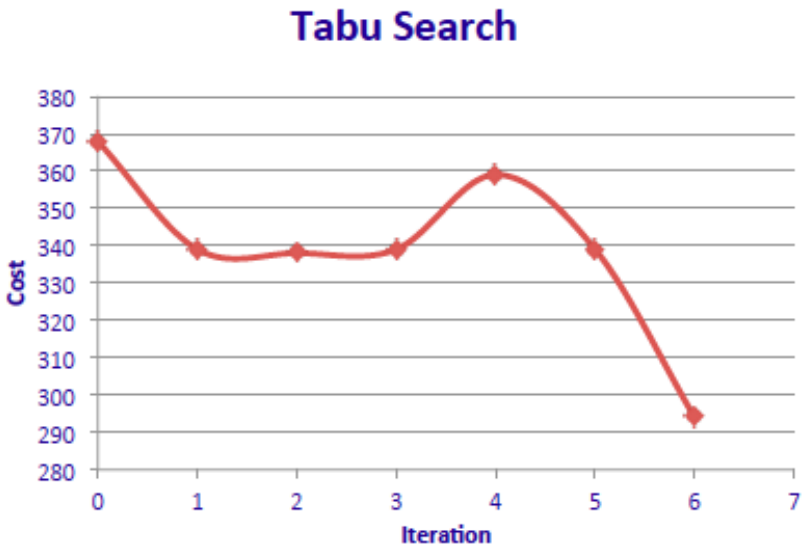
How can we avoid falling into cycles?

- Do not allow moves to solutions that we have seen before
- More generally: keep a tabu list
- We can block moves based on features of the solution
- Examples for TSP
  - If we make a relocation of city i, do not allow any move that concerns city i for the next 3 iterations
  - Every time we make a move (1-1 exchange or relocation) that affects city i, keep city i in the tabu list for the next k iterations
  - It's up to us to determine further conditions on what to keep in the tabu list for each iteration
- Very flexible way of blocking moves
- Also, very problem-specific method
- In practice, it works very well in discovering better local optima, instead of just stopping in the first local optimum

Tabu Search vs Classic Local Search

- A typical picture on the performance of tabu search, when compared against more traditional local search methods
- Comparison for TSP instances with 1-1 exchanges and relocations





Further Applications of Local Search

The MAX CUT problem

The MAX CUT Problem

MAX CUT:

I: An undirected graph G = (V,E) with nonnegative weights on its edges

Q: Find a cut, i.e., a partition (A, B) of V so as to maximize the total weight of the edges that cross the cut

Given a cut (A, B),
w(A, B) = sum of weights of edges crossing the cut

- Unlike minimum cut, MAX CUT is NP-complete
- Applications of MAX CUT: Circuit layout, statistical physics

The MAX CUT Problem

Designing local search algorithms for MAX CUT

- We should first think about defining the neighborhood of a solution

Single-flip neighborhood: Given a partition (A, B), a neighboring solution arises by moving one node from A to B, or one from B to A

- We can think of nodes in A as marked +1, and nodes in B as marked -1
- Improvement moves: say a node is improving if the cut becomes better when we move v to the other side of the partition

The MAX CUT Problem

1-flip-local-search algorithm

Start from a random partition (A, B)

While there exists an improving node

{ if v is in A, move v to B

else move v to A

}

Return (A, B)

Theorem:

The 1-flip local search algorithm achieves a 1/2-approximation for the MAX CUT problem

The MAX CUT Problem

- Unfortunately this is not a polynomial time algorithm
- It may require too many moves

Easy fix with very little loss:

- Make sure each move is improving "enough" the current solution
- Standard trick for other problems too

- Big-improvement-flip algorithm: Only choose a node which, when flipped, increases the cut value by at least $2\varepsilon/n\ w(A, B)$

Theorem:
The big-improvement flip algorithm achieves an approximation of $1/(2+\varepsilon)$

Proof very similar with the previous theorem

The MAX CUT Problem

Other approaches

- k-flip algorithm
  - Same as before but we now can flip up to k nodes at the same time
  - Expand neighborhood size to $O(n^k)$
- The Kernighan-Lin heuristic
  - Still uses a large neighborhood but avoids the computational overhead of performing k-flips
  - Very powerful in practice