Algorithms and Complexity

Coping with NP-hardness

Julián Mestre

School of Information Technologies
The University of Sydney

THE UNIVERSITY OF SYDNEY

So the problem is NP-hard, now what?

Imagine that your boss asks you to develop a piece of software to carry out a critical task in your company

After thinking about it for a while you realize that the problem is NP-hard, so you tell her so. But your boss is not impressed. She wants something in place to handle that critical task! What should you do? You may...

- exploit additional structure in your problem
- approximate your problem
- use a heuristic
- use fixed parameter algorithm
- model your problem as an integer program

Solving special cases

Graph problem that are NP-hard on general graphs, may be solvable in special graph classes, such as tress

The minimum weight vertex cover problem is the following:

- Input: graph G and a vertex weights $w : V \rightarrow Z^+$
- Task: Find a vertex cover S minimizing $w(S) = \sum_{u \text{ in } S} w(u)$

The problem is NP-hard as it is general that its unweighted version, which we already showed to be NP-hard!

Today, we will see how to solve this problem on trees.

DP for VC on trees

The key insight is if we remove a vertex we break the tree T into a number of subtrees, each defining an independent problem

Let T_u be the subtree rooted at u. Define DP states as follows:

- $L^{in}[u]$ = cost of vertex cover in T_u that uses u
- $L^{out}[u]$ = cost of vertex cover in T_u that doesn't use u

What is the recurrence for $L^{in}[u]$ and $L^{out}[u]$?

Where is the optimal solution for T?

Approximation algorithms

Another approach is to design algorithms that runs in polynomial time, but return solutions that are only close to the optimum

The minimum set cover problem is the following:

- Input: a collection $S_1, S_2, \ldots, S_m$ of subsets of a universal set U
- Task: Find a smallest sub-collection $C_1, \ldots, C_k$ such that $\cup C_i = U$

The problem is NP-hard: It generalizes minimum vertex cover

Can we at least find a set cover that has size close to the optimal?

Greedy algorithm

The algorithm works in iterations:

- In each iteration pick a set covering as many new elements as possible.
- Stop when all elements are covered

```
def Greedy(S):
  C = []
  U = union of sets in S
  while U not empty:
    next = set in S maximizing |next ∩ U|
    C.append(next)
    for e in next:
      U.remove(e)
  return C
```

Analysis of Greedy

Each chosen set in C sends a "bill" to its newly covered elements

For each set S in OPT, the "bills" sent to elements in S are at most

$$1 + 1/2 + 1/3 + \cdots + 1/|S| = H_{|S|}$$

Since OPT covers all elements, $|C| \leq H_n\, |OPT|$

Time complexity?

Thm.	Greedy is a poly-time H_n approximation for the minimum set cover problem

Minimum weighted set cover

The minimum weighted set cover problem is the following:

- Input: a collection $S_1, S_2, \ldots, S_m$ of subsets of a universal set U
- Each set S_i has associated a positive weight w_i
- Task: Find $C_1, \ldots, C_k$ minimizing $w_1 + \ldots + w_k$ such that $\cup\ C_i = U$

Although we can only get an approximately optimal solution, this is a very general problem that has many applications.

Thm. Greedy is a poly-time H_n approximation for the minimum weighted set cover problem

Local search

LS is an easy way of designing heuristics for optimization problems.

The main ingredients are:

- C: a set of feasible solutions
- f: a cost function
- way of choosing initial solution
- neighborhood function

```
def local_search(C, neighborhood, f):
  // usually C is given implicitly

  X = select initial solution from C
  while True:
    Y = solution in neighborhood(X)
        minimizing f(Y)
    if f(Y) < f(X):
      X = Y
    else:
       break
  return X
```

LS can be used for maximization problems as well

Maximum cut

The maximum cut problem is the following:

- Input: undirected graph $G=(V,E)$ and edge capacities $c : E \rightarrow Z^+$
- Task: Find a cut (A,B) maximizing $c(A,B)$

Ingredients for local search algorithm

- Feasible solutions: All possible cuts (A,B)
- Initial solution: Random cut
- Cost function: $f(A,B) = c(A,B)$
- Neighboring function: Flip a node from one size of the cut to the other side

Local search for Max cut

In the k-flip neighborhood, k vertices are allowed to change sides

Quality of local optima:

- Flip and k-Flip: local optima are 0.5-approximate
- Flip and k-Flip: there are example attaining this bound
- k-Flip yields better results in practice

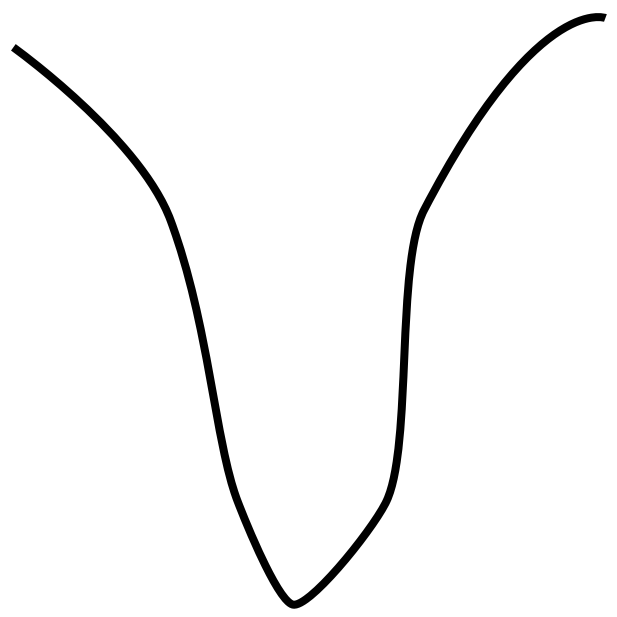
Time complexity (finding a good neighboring solution)

- Flip: Each solution has n neighbors, so it takes $O(n\ m)$ time
- k-Flip: Each solution has $\Theta(n^k)$ neighbors, so ti takes $O(n^k\ m)$

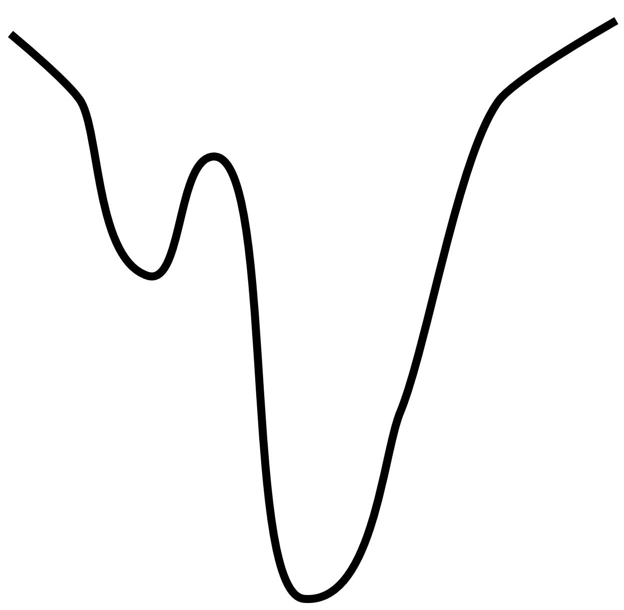
The Kenighan-Li neighborhood is in between these two extremes

Landscape of optimization problems

Physical systems tend toward low energy configurations. We can think of a local search algorithm as trying to reach a local minimum defined by the potential energy f



Amenable to
local search



Not amenable to
local search

Local vs global
minima

Metropolis algorithm

Main idea:

- perform local search
- with some probability allow moves to solutions that do not improve objective
- where $p(X, Y, T) = \exp(-(f(X)-f(Y)) / k T)$ here T and k are parameters

Depending on T

- always jump if T is large
- never jump if $T = 0$

```
// In each iteration do the following

X = current solution
Y = neighbor of X chosen at random
if f(Y) < f(X):
  X = Y
else:
  with probability p(X,Y,T) set X = Y
```

Metropolis and simulated annealing

Let $Z = \sum_X \exp(-f(X)/kT)$, then the fraction of the time Metropolis spends on state X during the first t steps tends to

$\exp(-f(X) / kT) / Z$

No guaranteed on how quickly we reach this steady state

as t tends to infinity

Simulated annealing uses the Metropolis algorithm but varies the parameter T as the algorithm progresses.

- Initially T is very large (allowing wild jumps)
- Progressively T is reduced

Fixed parameter tractable

Suppose you wanted to solve an instance of the minimum vertex cover problem on a graph with n vertices where you know the size of the minimum vertex cover to be k

You could try to enumerate all subsets of vertices of size k, but that would run in $\Omega(n^k)$ time. This is useless for small instances like $n=1000$ and $k = 10$. Can we do better than that?

Yes! Using branching we can solve the problem in $O(2^k\, n)$ time. Therefore, the problem is tractable for very small values of k even if the graph is very large!

Branching rules

<u>Obs.</u>: Suppose that G has a vertex cover of size k. For all edges (u,v) in G either G-u or G-v has a vertex cover of size k-1.

<u>Obs.</u>: For some edge (u,v) in G if neither G-u or G-v has a vertex cover of size k-1, then G doesn't have a vertex cover of size k

```
def branching(G=(V,E),k):
  if |E| = 0:
    return empty set
  else if k = 0:
    return "No VC of size k"
  else:
     (u,v) = some edge in E
     C = branching(G-u, k-1)
     if C is a VC for G-u:
       return C + u
     C = branching(G-v, k-1)
     if C is a cover for G-v:
       return C + v
     return "No VC of size k"
```

Time complexity

Let $T(n,k)$ be the time complexity of branching on a graph with n vertices and target vertex cover size k. Then

$$T(n,k) \leq 2\,T(n-1, k-1) + O(n)$$

You need to pass the graph by value and be careful to "reconstitute" the graph after removing u and v

It follows that $T(n,k) = O(2^k\, n)$

Integer Programs

The three main ingredients of an integer program are

- Variables: can take discrete values, say $x_1, x_2, \ldots, x_n$ in $\{0,1\}$
- Constraints: must be linear inequalities, say $2 x_1 - x_2 >= 10$
- Objective function: must also be linear, say $2 x_1 + 3 x_2$

How should we set the variables so that all constraints are obeyed and the objective is maximum/minimum?

The problem is NP-hard but there are good solvers that can handle fairly large instances.

Most of the work goes into modeling your problem as an IP.

Coping with NP hardness

Do not give up if you need to solve a problem that is NP-hard!

You can try to:

- Exploit some special property of your instance
- Use an approximation algorithm
- Use heuristic method
- Fixed parameter tractable algorithm
- Integer programming or similar