DPV Chapter 9

Dealing with NP-Completeness

Jim Royer

April 17, 2019

Uncredited diagrams are from DPV or homemade.

So the problem you want to solve is NP-Complete…

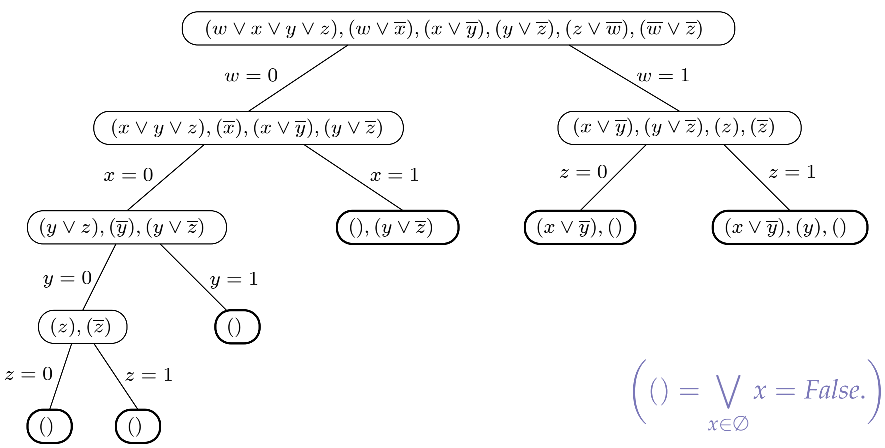
Now what?

- ✘ Give up.
- ✘ Burn cycles and try to solve it exactly.
- ✘ Try the first thing that comes into your head and hope it produces correct answers and is fast enough to get by.
- ✔ Open a different tool box. (Chapter 9 of DPV.)

Backtracking = exhaustive search + pruning

Example: SAT via Backtracking

Let $\varphi = (w \vee x \vee y \vee z) \wedge (w \vee \overline{x}) \wedge (x \vee \overline{y}) \wedge (y \vee \overline{z}) \wedge (z \vee \overline{w}) \wedge (\overline{w} \vee \overline{z})$.



$\left(() = \bigvee_{x \in \emptyset} x = \mathit{False}. \right)$

Backtracking: The general scheme

First we need a fast test for subproblems such that

$$test(P) = \begin{cases} \text{failure}, & \text{if subproblem } P \text{ has no solution;} \\ \text{success}, & \text{if a solution to } P \text{ is found;} \\ \text{uncertainty}, & \text{otherwise.} \end{cases}$$

Then:

```
Start with some problem P0
S ← { P0 } // the set of active subproblems
while (S ≠ ∅) do
   Choose a P ∈ S;  S ← S − { P }
   Expand P into subproblems P1, ..., Pk
   for i ← 1 to k do
      case test(Pi) of
         success: announce solution and halt
         failure: discard Pi
         uncertainty: add Pi to S
Announce that there is no solution.
```

For SAT:

- *Choose* ≡ pick a clause
- *Expand* ≡ pick a variable in the clause

Branch-and-Bound

- B&B = the backtracking idea for optimization problems
- We consider minimization problems.
- First we need a fast way to compute *lower bounds* for the cost.
- Then:

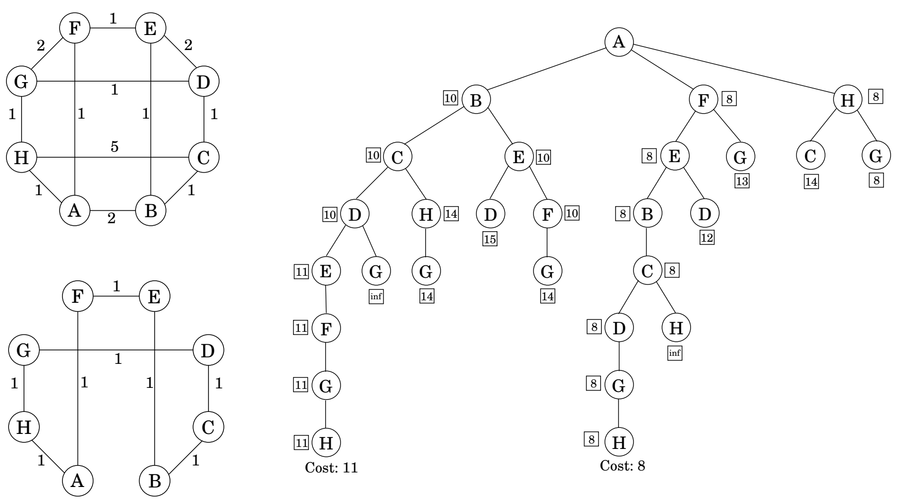
```
Start with some problem P0
S ← { P0 } // the set of active subproblems
bestSoFar ← ∞
while (S ≠ ∅) do
   Choose a P ∈ S;  S ← S − { P }
   Expand P into subproblems P1, ..., Pk
   for i ← 1 to k do
      if (Pi is a complete solution) then update bestSoFar
      else if (lowerbound(Pi) < bestSoFar) then add Pi to S
return bestSoFar
```


Branch-and-Bound Applied to TSP, 1

- $G = (V, E)$ each $e \in E$ with length $d_e > 0$.
- Fix an $a \in V$.
- Partial solution: $[a, S, b]$ = a path from a to b, S = the verts in this path
- Initial subproblem: $[a, \{ a \}, a]$.
- Extension: $[a, S \cup \{ x \}, x]$ where $x \in (V - S)$ and $(b, x) \in E$.
- $lowerbound([a, S, b])$
  = a lower bound on the cost of completing the partial tour $[a, S, b]$
  = the sum of:
  + the cheapest edge from a to $V - S$.
  + the cheapest edge from b to $V - S$.
  + the cost of a minimum spanning tree of $V - S$.

?? Why is this a lower bound on the cost of completing the partial tour $[a, S, b]$?

Branch-and-Bound Applied to TSP, 2



- 28 partial solutions examined.
- $7! = 5,040$ partial solutions in a brute-force search.

Approximation Algorithms

- Instead of seeking an optimum solution, try "close to optimum"
- The question is how close is good enough.
- $opt(I)$ = the value of an optimum solution for instance I.
- **Convention:** Assume $opt(I)$ is always a positive integer.
- **Convention:** Focus on **min**imization problems.
- Suppose $\mathcal{A}(I)$ is the solution for I an algorithm $\mathcal{A}$ returns.
- The *approximation ratio for* $\mathcal{A}$ is

$$\alpha_{\mathcal{A}} \quad = \quad \max_{I} \frac{\mathcal{A}(I)}{Opt(I)} \quad \geq \quad 1.$$

- For **max**imization problems, take:

$$\alpha_{\mathcal{A}} \quad = \quad \max_{I} \frac{Opt(I)}{\mathcal{A}(I)} \quad \geq \quad 1.$$

- (The closer $\alpha_{\mathcal{A}}$ is to 1 the better.)

Recall from Chapter 5: *Set Cover, 1*

Suppose B is a set and $S_1, \ldots, S_m \subseteq B$.

Definition

(a) A set cover of B is a $\{ S'_1, \ldots, S'_k \} \subseteq \{ S_1, \ldots, S_m \}$ with $B \subseteq \cup_{i=1}^{k} S'_i$

(b) A minimal set cover of B is a set cover of B using as few of the S_i-sets as possible.

The Set Cover Problem (SCP)

Given: B and $S_1, \ldots, S_m$ as above.
Find: A minimal set cover of B.

Example

For: $B = \{ 1, \ldots, 14 \}$ and

$$
\begin{aligned}
S_1 &= \{ 1, 2 \} \\
S_2 &= \{ 3, 4, 5, 6 \} \\
S_3 &= \{ 7, 8, 9, 10, 11, 12, 13, 14 \} \\
S_4 &= \{ 1, 3, 5, 7, 9, 11, 13 \} \\
S_5 &= \{ 2, 4, 6, 8, 10, 12, 14 \}
\end{aligned}
$$

the solution to SCP is $\{ S_4, S_5 \}$.

Recall from Chapter 5: *Set Cover, 2*

A Greedy Approximation to the Set Cover Problem

```
// Input: B and S_1, ..., S_m ⊆ B as above.
// Output: A set cover of B which is close to minimal.
C ← ∅
while (some element of B is not yet covered) do
   Pick the S_i with the largest number of uncovered B-elements
   C ← C ∪ { S_i }
return C
```

Example

$$
\begin{aligned}
B &= \{ 1, \ldots, 14 \} \\
S_1 &= \{ 1, 2 \} \\
S_2 &= \{ 3, 4, 5, 6 \} \\
S_3 &= \{ 7, 8, 9, 10, 11, 12, 13, 14 \} \\
S_4 &= \{ 1, 3, 5, 7, 9, 11, 13 \} \\
S_5 &= \{ 2, 4, 6, 8, 10, 12, 14 \}
\end{aligned}
$$

On this, the algorithm returns $\{ S_1, S_2, S_3 \}$.

Recall from Chapter 5: *Set Cover, 3*

A Greedy Approx. to SCP

```
// Input: B and S_1, ..., S_m ⊆ B
// Output: A near min. set cover
C ← ∅
while (all of B is not covered) do
   Pick the S_i with the largest
      number of uncovered B-elms
   C ← C ∪ { S_i }
return C
```

Claim

Suppose B contains n elements and the min. cover has k sets.

Then the greedy algorithm will use at most $k \log_e n$ sets.

Proof: Let
n_t = the number of uncovered elms after t-many while loop iterations

So $n_0 = n$.
After iteration t:

- there are n_t elms left.
- k many sets cover them
- So there must be some set with at least n_t/k many elements.
- So by the greedy choice,

$$
\begin{aligned}
n_{t+1} \le n_t - \frac{n_t}{k} &= n_t \left(1 - \frac{1}{k}\right) \\
&= n_0 \left(1 - \frac{1}{k}\right)^t .
\end{aligned}
$$

Recall from Chapter 5: *Set Cover, 4*

A Greedy Approx. to SCP

```
// Input: B and S_1, ..., S_m ⊆ B
// Output: A near min. set cover
C ← ∅
while (all of B is not covered) do
   Pick the S_i with the largest
      number of uncovered B-elms
   C ← C ∪ { S_i }
return C
```

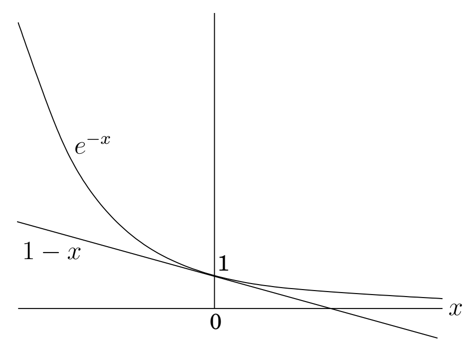
Claim

Suppose B contains n elements and the min. cover has k sets.
Then the greedy algorithm will use at most $k \log_e n$ sets.

Proof: Let
n_t = the number of uncovered elms after t-many while loop iterations

We know: $n_{t+1} \le n \left(1 - \frac{1}{k}\right)^t$.

Fact: $1 - x \le e^{-x}$ for all x, with equality iff $x = 0$.



Recall from Chapter 5: *Set Cover, 5*

A Greedy Approx. to SCP

```
// Input: B and S_1, ..., S_m ⊆ B
// Output: A near min. set cover
C ← ∅
while (all of B is not covered) do
    Pick the S_i with the largest
      number of uncovered B-elms
    C ← C ∪ { S_i }
return C
```

Claim

Suppose B contains n elements and the min. cover has k sets. **Then** the greedy algorithm will use at most $k \log_e n$ sets.

Proof: Let
n_t = the number of uncovered elms after t-many while loop iterations

We know: $n_{t+1} \leq n\left(1 - \frac{1}{k}\right)^t$.

Fact: $1 - x \leq e^{-x}$ for all x, with equality iff $x = 0$.

$\therefore$ At $t \geq k \log_e n$, $n_t < ne^{-\log_e n} = 1$, i.e., we must have covered all of B.

So the greedy algorithm is optimal within a $\log_e n$ factor.

That is,

$$\alpha_{\mathcal{A}} = \max_I \frac{\mathcal{A}(I)}{Opt(I)} \leq \log_e n.$$

Approximating Vertex Cover, 1

Vertex Cover (as an optimization problem)

Given: $G = (V, E)$ an undirected graph
Find: $S \subseteq V$ such that S touches every edge.
Goal: Minimize $|S|$.

- Vertex Cover is a special case of Set Cover.
- Therefore, it can be approximated within a $O(\log n)$ factor.
- *However*, it turns out we can do much better.

Approximating Vertex Cover, 2

Definition

Suppose $G = (V, E)$ an undirected graph.

(a) A *matching* is an $M \subseteq E$ such that any two edges in M have no endpoints in common.

(b) M is a *maximum matching* when for each $e \in (E - M)$, $M \cup \{ e \}$ fails to be a matching.

Observations

- Maximal matchings are easy to construct. *(How?)*
- Fix G.
- If C is a vertex cover and M is a maximum matching, then each $(u, v) \in M$ must have at least one of u and v in C. *(Why?)*

$\therefore$ (the size of a min. vertex cover for G) $\geq$ (the size of a max. matching for G)

- If M is a maximal matching, then $S = \{ u \mid u$ is an endpoint of an $e \in M \}$ is a vertex cover. *(Why?)*

$\therefore$ $|S| = 2|M| \geq$ (the size of a min. vertex cover for G) $\geq |M|$.

Approximating Vertex Cover, 3

An approximation algorithm for Vertex Cover

```
input G = (V, E)
Find a maximal matching M ⊆ E.
return S = { u | u is an endpoint of an e ∈ M }
```

- By the Observations, the approximation ratio of this algorithm is $\alpha_{\mathcal{A}} \leq 2$.
- In fact, you can find examples where the ratio is exactly 2.

$\therefore$ The approximation ratio of *this* algorithm is $\alpha_{\mathcal{A}} = 2$.

- *What about other algorithms?*

Amazing Fact (Dinur and Safra, 2005)

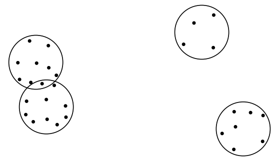
Minimum vertex cover cannot be approximated within a factor of 1.3606 for any sufficiently large vertex degree unless P=NP.

Clustering, 1

Definition

A *metric* on a space X is a function $d\colon X \times X \to \mathbb{R}^{\geq 0}$ such that, for all $x, y, z \in X$:

1. $d(x,y) \geq 0$.
2. $d(x,y) = 0 \implies x = y$.
3. $d(x,y) = d(y,x)$.
4. $d(x,y) \leq d(x,z) + d(z,y)$.



data points/four clusters

k-Clustering

Input: Points $X = \{ x_1, \dots, x_n \}$, metric d, integer $k > 0$.
Output: A partition of X into k clusters $C_1, \dots, C_k$.
Goal: Minimize the diameter of the clusters: $\max_j \max_{x,x' \in C_j} d(x,x')$.

- k-Clustering is NP-complete.
- k-Clustering is important in lots of areas (e.g., data mining). See http://en.wikipedia.org/wiki/K-means_clustering

Clustering, 2

Approximation Algorithm for k-Clustering

Pick any point $p_1 \in X$ to start
for $i \leftarrow 2$ **to** k **do**
  $p_i \leftarrow$ a point in X that is farthest away from $p_1, \dots, p_{i-1}$
    // I.e., p_i maximizes: $\min\{ d(\cdot, p_j) : j = 1, \dots, i-1 \}$
Create k clusters: $C_i = \{ x \in X : p_i$ is the closest center $\}$

Claim: For the above algorithm, $\alpha_{\mathcal{A}} \leq 2$.

Proof:

- Let x be the point farthest from $p_1, \dots, p_k$.
- Let r = the distance of x to the nearest p_i.
- ∴ Every point must be within r from its cluster center.
- ∴ The diameter of the clusters is $\leq 2r$.
- The points $p_1, \dots, p_k$ and x are all $\geq r$ distant from one another.
- Any partition of X into k cluster must put two of $p_1, \dots, p_k, x$ into the same cluster. (By the PHP.)
- ∴ These clusters must have diameter $\geq r$. QED

Traveling Salesman with metric distances, 1

Traveling Salesman Problem

Given: n vertices and all $n \cdot (n-1)/2$-many distances between them.
Find: An ordering of $1, \dots, n$: $\pi(1), \pi(2), \dots, \pi(n)$ so that the tour's cost $d(\pi(1), \pi(2)) + d(\pi(2), \pi(3)) + \cdots + d(\pi(n), \pi(1))$ is minimal.

Question: Suppose we require the distances to come from a metric. Does this help make the problem easier? Answer: Yes!

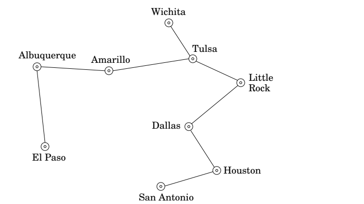
Definition (Repeated)

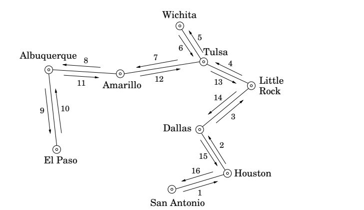
A *metric* on a space X is a function $d\colon X \times X \to \mathbb{R}^{\geq 0}$ such that, for all $x, y, z \in X$:

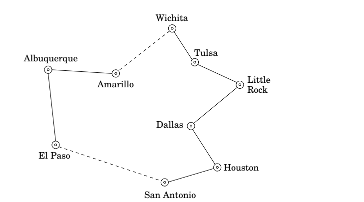
1. $d(x,y) \geq 0$.
2. $d(x,y) = 0 \implies x = y$.
3. $d(x,y) = d(y,x)$.
4. $d(x,y) \leq d(x,z) + d(z,y)$.

Traveling Salesman with metric distances, 2

- Take a TSP path and delete an edge. The result is a spanning tree.
- ∴ (cost of a MST for G) < (cost of a solutions to TSP for G)
- Now take T, a MST for G. Turn T into a tour that uses each edge twice.
- Let $c_1, \dots, c_n$ be the cities on the tour — *in the order they are first visited*.
- Edit the tour so that from city c_i the tour shortcuts to city c_{i+1} and from city c_n it shortcuts to city c_1.
- By the triangle inequality, the shortcuts can keep the cost the same or improve it.
- ∴ (cost of a solutions to TSP for G) < 2 × (cost of a MST for G)
- ∴ We can approximate the metric version of TSP within a factor of 2.







RECALL: Rudrata/Hamiltonian Cycle $\trianglelefteq$ TSP

Rudrata/Hamiltonian Cycle Problem

Given: $G = (V, E)$, an undirected graph.
Find: A simple cycle that visits each vertex of G.

Traveling Salesman Problem (TSP)

Given: V', n vertices; all $\frac{n\cdot(n-1)}{2}$-many distances between them; and b, a budget
Find: π, an ordering of $1, \dots, n$, such that $\sum_{i=1}^{n} d_{\pi(i),\pi(1+(i \bmod n))} \leq b$.

Construction of $I(G, C)$.
Given $G = (V, E)$ and $C \geq 1$, define

$$V' = V$$

$$d_{i,j} = \begin{cases} 1, & \text{if } (i,j) \in E; \\ 1 + C, & \text{otherwise.} \end{cases}$$

$$b = |V|$$

Claim: (V, E) has a R/H cycle
$\iff$ (V', d) has tour of cost $\leq b$.

If $C \gg 1$:

- Gap: either a solution of cost n, or solutions with costs $\geq n + C$, but none inbetween.

$\therefore$ An approx. solution to (the full) TSP would let us solve Ham. Cycle in polytime!
How? (See next page.)

Approximating General TSP

Claim

An approximate solution to TSP would give us polytime solution of Rudrata Path.

Proof

- Suppose that we had $\mathcal{A}$, a polytime approximation algorithm for TSP with approximation factor $\alpha_{\mathcal{A}}$.
- Suppose G is any instance of Rudrata Path.
- Construct $I(G, C)$ where $C = \alpha_{\mathcal{A}} \cdot n$ and run $\mathcal{A}$ on it.
- If G has a Rudrata path, then $OPT(I(G, C)) = n$ and $\mathcal{A}$ finds a TSP tour of cost $\alpha_{\mathcal{A}} \cdot OPT(I(G, C)) = \alpha_{\mathcal{A}} \cdot n$.
- If G has no Rudrata path, then $\mathcal{A}$ must return a tour of cost $> \alpha_{\mathcal{A}} \cdot n$.
- Since $\mathcal{A}$ is supposed to run in polytime, this means we can decide Rudrata path in polytime!!!!

Corollary

- *If TSP has a polytime approximation algorithm, then P=NP.*
- *If P$\neq$NP, then TSP has no polytime approximation algorithm.*

Approximating Knapsack, 1

Knapsack without repetition

Given:

- A knapsack with capacity W.
- Items $1, \dots, n$
- Item i has weight w_i & value v_i.

Find: a set $M \subseteq \{1, \dots, n\} \ni$

- $\sum_{i \in M} w_i \leq W$ and
- $\sum_{i \in M} v_i$ is maximized.

- By Chapter 6, there is a dynamic programming solution to Knapsack that runs in $O(n \cdot W) = O(n \cdot 2^{|W|})$ time.
- There is a similar dynamic programming solution to Knapsack that runs in $O(n \cdot V) = O(n \cdot 2^{|V|})$ time, where $V = \sum_{i=1}^{n} v_i$.
- We use the $O(n \cdot V)$ version as the basis for an approximation algorithm.

Approximating Knapsack, 2

```
function ksApprox(v⃗, w⃗, W, ε)    // ε = an approximation factor
   // Assume each w_i ≤ W.
   v_max ← max{ v_i : i = 1,...,n }.
   for i = 1,...,n do v̂_i ← ⌊ (v_i · n) / (v_max · ε) ⌋.    // Rescale the values
   Run the dynamic programming algorithm using the v̂_i values.
   return the resulting choices of items
```

Runtime Analysis

- Since each $\hat{v}_i \leq n/\epsilon$, we have $\hat{v}_1 + \cdots + \hat{v}_n \leq n^2/\epsilon$.
- So the DP algorithm runs in $O(n^3/\epsilon)$ time.

Approximating Knapsack, 3

```
function ksApprox(v⃗, w⃗, W, ε)    // ε = an approximation factor
  // Assume each w_i ≤ W.
  v_max ← max{ v_i : i = 1,...,n }.
  for i = 1,...,n do v̂_i ← ⌊ v_i · n / (v_max · ε) ⌋.    // Rescale the values
  Run the dynamic programming algorithm using the v̂_i values.
  return the resulting choices of items
```

Approximation Analysis Suppose:

- S is an optimal solution to the original problem with total value K^*.
- $\hat{S}$ is the solution produces for the scaled problem.

Then: $$\sum_{i\in S} \hat{v}_i = \sum_{i\in S} \left\lfloor \frac{v_i \cdot n}{v_{max} \cdot \epsilon} \right\rfloor \geq \sum_{i\in S} \left(\frac{v_i \cdot n}{v_{max} \cdot \epsilon} - 1 \right) = K^* \cdot \frac{n}{v_{max} \cdot \epsilon} - n.$$

So, the value of $\hat{S}$ is at least $K^* \cdot \frac{n}{v_{max}\cdot\epsilon} - n$. Hence,

Correction: The boxed part is what I missed in class.

$$\sum_{i\in \hat{S}} v_i \geq \frac{v_{max}\cdot\epsilon}{n} \sum_{i\in \hat{S}} \hat{v}_i \geq \frac{v_{max}\cdot\epsilon}{n} \left(K^* \cdot \frac{n}{v_{max}\cdot\epsilon} - n \right) = K^* - v_{max}\cdot\epsilon \geq K^*(1-\epsilon).$$

The approximability hierarchy

- No finite approximation ratio is possible.
  E.g., TSP.
- An approximation ratio of about $\log n$ is possible.
  E.g., Set Cover.
- A constant approximation ratio is possible,
  but there are limits to how small this can be.
  E.g., Vertex Cover, k-Clustering, and metric TSP.
  The proofs of these lower limit results are ***really hard!!!***
- A constant approximation ratio is possible,
  and in fact you can make $\alpha_{\mathcal{A}}$ arbitrarily close to 1.
  E.g., Knapsack.

NOTE: All of the above assumes P≠NP.

- If P=NP, all the problems can be solved exactly in polytime.

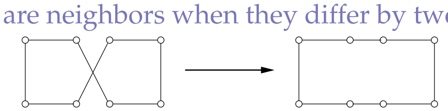
Local search heuristics: The general scheme

```
s ← any initial solution
while there is a solution s' in the neighborhood of s with cost(s') < cost(s) do
  s ← s'
return s
```

For any application of this scheme to a particular problem,
the key question *what is a good notion of neighborhood?*

Local search heuristics: Traveling Salesman, 1

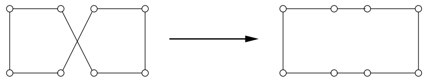
- Assume we have a complete graph on n vertices
  (with a cost assigned to each edge).
- So there are $(n-1)!$ many tours.
- Two tours differ by at least two edges. *(Why?)*
- So let us try:
  Tours T_1 and T_2 are neighbors when they differ by two edges.



- With this choice of "neighbor":
  1. ***What is the overall running time?***
  2. ***Does this always return an optimal answer?***
- Answers:
  1. Hard to say.
  2. Of course not.

Local search heuristics: Traveling Salesman, 2

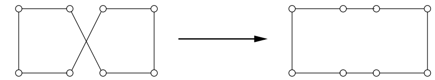
- Tours T_1 and T_2 are neighbors when they differ by two edges.



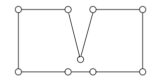
- With this choice of "neighbor":
  What is the overall running time?
  - Each tour has $O(n^2)$ neighbors,
    so making the choice is not too expensive.
  - But, the algorithm may well go through exponentially many iterations.

Local search heuristics: Traveling Salesman, 3

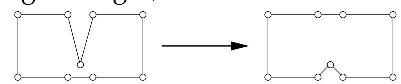
- Tours T_1 and T_2 are neighbors when they differ by two edges.



- With this choice of "neighbor":
  Does this always return an optimal answer?
  - The final answer will be ***locally optimal***, but not necessarily optimal.
  - The problem is that this notion of neighbor is too myopic. E.g.,

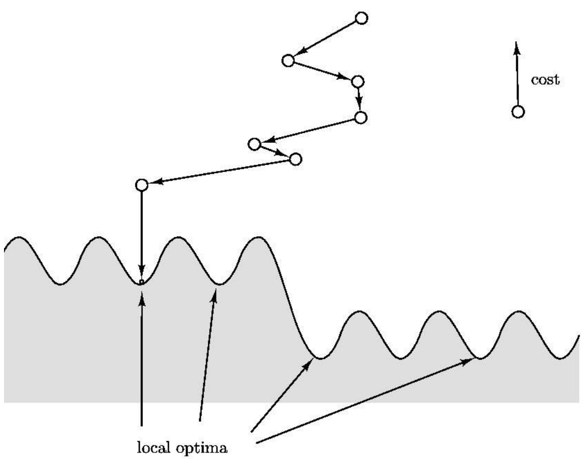


- If we allow three-edge changes, then:



  but then a tour has $O(n^3)$ neighbors and the choice part of the algorithm slows down.

Local search heuristics: Optima, Local vs. global



Local search: Graph partitioning, 1

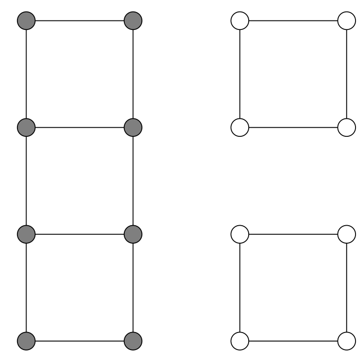
Graph partitioning

Given: $G = (V, E)$, an undirected graph with nonnegative edge wghts, and $\alpha \in (0, 1/2]$.
Return: A partition of V into A and B with

$$|A|, |B| \geq \alpha|V|.$$

Goal: Minimize the capacity of the (A, B)-cut.

Note: The general problem is reducible to the special case of $\alpha = 1/2$.



Strategy:

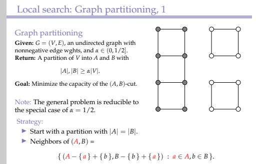
- Start with a partition with $|A| = |B|$.
- Neighbors of (A, B) =

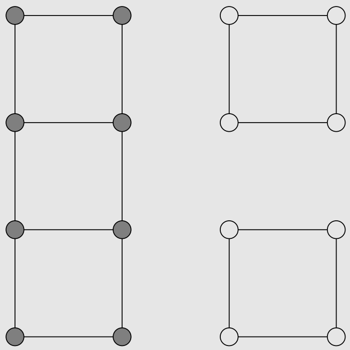
$$\{\, (A - \{\, a \,\} + \{\, b \,\}, B - \{\, b \,\} + \{\, a \,\}) \; : \; a \in A, b \in B \,\}.$$

Dealing with NP-Completeness

2019-04-17

└─Local search: Graph partitioning, 1



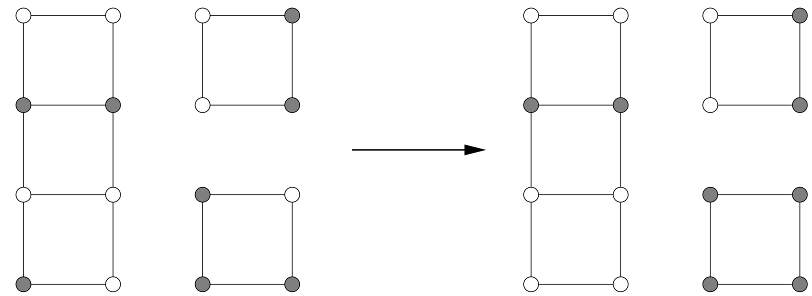


- A = gray verts, B = white verts.
- Weights 0 and 1
- Optimal partition as cost 0.

Local search: Graph partitioning, 2

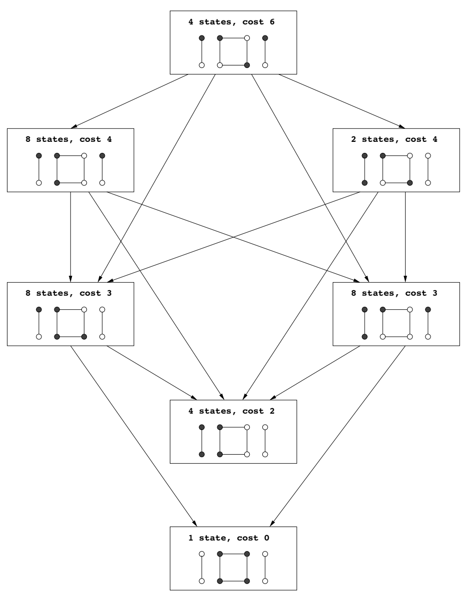
- Start with a partition with $|A| = |B|$.
- Neighbors of (A, B) =

$$\{ (A - \{ a \} + \{ b \}, B - \{ b \} + \{ a \}) \ : \ a \in A, b \in B \}.$$



Local search: Graph partitioning, 3

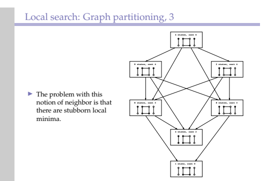
- The problem with this notion of neighbor is that there are stubborn local minima.

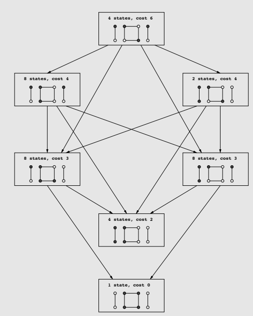


Dealing with NP-Completeness

2019-04-17

└─Local search: Graph partitioning, 3





- Search space for a graph with 8 nodes.
- Then entire space has 35 solutions, but the picture has grouped these into seven groups to cut the clutter.
- There are five local optima.

Dealing with local optima: Randomized Restarts

$L \leftarrow$ an empty list
repeat k times
  $s \leftarrow$ a randomly chosen initial solution
  while (there is a solution s' in the neighborhood of s with $cost(s') < cost(s)$) **do**
    $s \leftarrow s'$
  add s to L
end-repeat
return the best solution in L

This can shake free of bad local optima.

Dealing with local optima: Simulated Annealing

$s \leftarrow$ a randomly chosen initial solution
repeat
  $s' \leftarrow$ a randomly chosen solution in the neighborhood of s
  $\Delta \leftarrow cost(s') - cost(s)$
  if $(\Delta < 0)$ **then** $s \leftarrow s'$
  else with probability $e^{-\Delta/T}$ **do** $s \leftarrow s'$
until we decide we are done

- $T \equiv$ temperature
- If $T \approx 0$ this is roughly the previous scheme.
- If T is big, then s jumps around a lot.
- We vary T, initially large (hot), and gradually small (cooler).