Chapter 8

Approximation Algorithms

Outline

- Why approximation algorithms?
- The vertex cover problem
- The set cover problem
- TSP

Why Approximation Algorithms ?

- Many problems of practical significance are NP-complete but are too important to abandon merely because obtaining an optimal solution is intractable.
- If a problem is NP-complete, we are unlikely to find a polynomial time algorithm for solving it exactly, but it may still be possible to find *near-optimal solution* in polynomial time.
- In practice, near-optimality is often good enough.
- An algorithm that returns near-optimal solutions is called an *approximation algorithm.*

Performance bounds for approximation algorithms

- i is an optimization problem instance
- $c(i)$ be the cost of solution produced by approximate algorithm and $c^*(i)$ be the cost of optimal solution.
- For minimization problem, we want $c(i)/c^*(i)$ to be as small as possible.
- For maximization problem, we want $c^*(i)/c(i)$ to be as small as possible.
- An approximation algorithm for the given problem instance i, has a ratio bound of $p(n)$ if for any input of size n, the cost c of the solution produced by the approximation algorithm is within a factor of $p(n)$ of the cost c* of an optimal solution. That is

$$\max(c(i)/c^*(i),\ c^*(i)/c(i)) \le p(n)$$

- Note that $p(n)$ is always greater than or equal to 1.
- If $p(n) = 1$ then the approximate algorithm is an optimal algorithm.
- The larger $p(n)$, the worst algorithm
- **Relative error**
  - We define the *relative error* of the approximate algorithm for any input size as

$$|c(i) - c^*(i)| / c^*(i)$$

  - We say that an approximate algorithm has a **relative error bound** of $\varepsilon(n)$ if

$$|c(i)-c^*(i)|/c^*(i) \leq \varepsilon(n)$$

1. The Vertex-Cover Problem

- Vertex cover: given an undirected graph $G=(V,E)$, then a subset $V' \subseteq V$ such that if $(u,v) \in E$, then $u \in V'$ or $v \in V'$ (or both).
- Size of a vertex cover: the number of vertices in it.
- Vertex-cover problem: find a vertex-cover of minimal size.
- This problem is NP-hard, since the related decision problem is NP-complete

Approximate vertex-cover algorithm

APPROX-VERTEX-COVER(G)

$C \leftarrow \emptyset$
$E' \leftarrow E[G]$
while $E' \neq \emptyset$
    do let (u, v) be an arbitrary edge of E'
        $C \leftarrow C \cup \{u, v\}$
        remove from E' every edge incident on either u or v
return C

The running time of this algorithm is O(E).

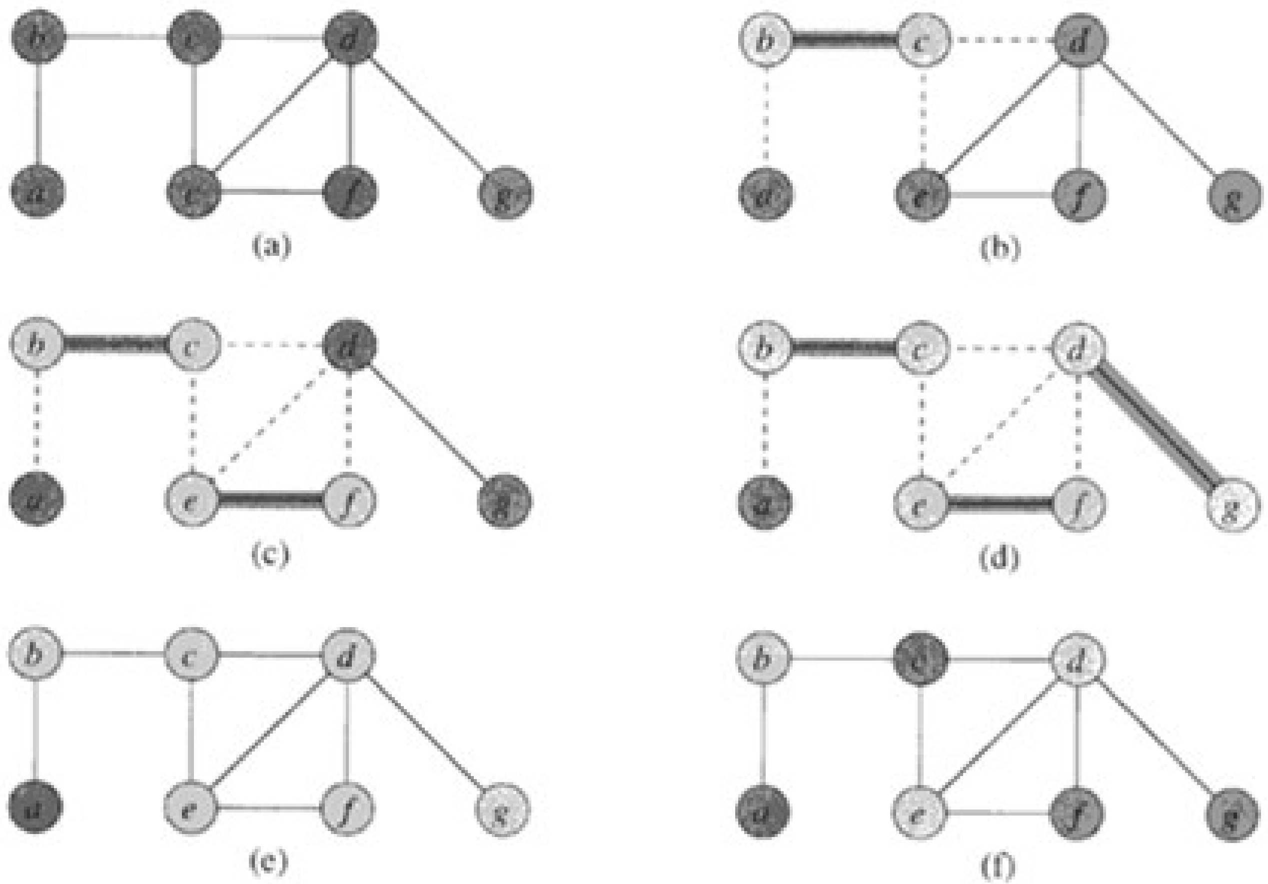


Figure 35.1 The operation of APPROX-VERTEX-COVER. **(a)** The input graph G, which has 7 vertices and 8 edges. **(b)** The edge (b, c), shown heavy, is the first edge chosen by APPROX-VERTEX-COVER. Vertices b and c, shown lightly shaded, are added to the set C containing the vertex cover being created. Edges (a, b), (c, e), and (c, d), shown dashed, are removed since they are now covered by some vertex in C. **(c)** Edge (e, f) is chosen; vertices e and f are added to C. **(d)** Edge (d, g) is chosen; vertices d and g are added to C. **(e)** The set C, which is the vertex cover produced by APPROX-VERTEX-COVER, contains the six vertices b, c, d, e, f, g. **(f)** The optimal vertex cover for this problem contains only three vertices: b, d, and e.

Theorem:

- APPROXIMATE-VERTEX-COVER has a ratio bound of 2, i.e., the size of returned vertex cover set is at most twice of the size of optimal vertex-cover.
- Proof:
  - It runs in poly time
  - The returned C is a vertex-cover.
  - Let A be the set of edges picked in line 4 and C* be the optimal vertex-cover.
    - Then C* must include at least one end of each edge in A and no two edges in A are covered by the same vertex in C*, so $|C^*| \geq |A|$.
    - Moreover, $|C| = 2|A|$, so $|C| \leq 2|C^*|$.

The Set Covering Problem

- The set covering problem is an optimization problem that models many resource-selection problems.
- An instance (X, F) of the set-covering problem consists of a finite set X and a family F of subsets of X, such that every element of X belongs to at least one subset in F:

$$X = \bigcup_{S \in F} S$$

We say that a subset $S \in F$ *covers* its elements.

- The problem is to find a minimum-size subset $C \subseteq F$ whose members cover all of X:

$$X = \bigcup_{S \in C} S$$

- We say that any C satisfying the above equation covers X.

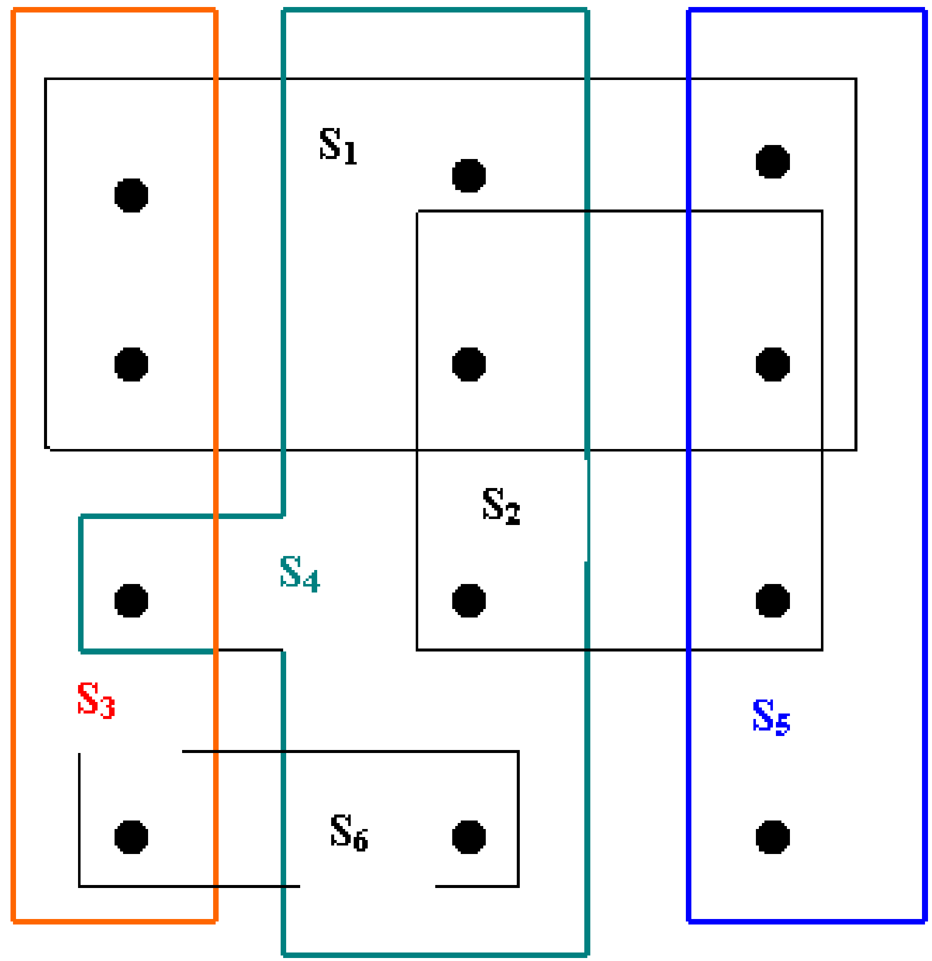


Figure 6.2 An instance {X, F} of the set covering problem, where X consists of the 12 black points and F = { S_1, S_2, S_3, S_4, S_5, S_6}. A minimum size set cover is C = { S_3, S_4, S_5}. The greedy algorithm produces the set C' = {S_1, S_4, S_5, S_3} in order.

Applications of Set-covering problem

- Assume that X is a set of skills that are needed to solve a problem and we have a set of people available to work on it. We wish to form a team, containing as few people as possible, s.t. for every requisite skill in X, there is a member in the team having that skill.
- Assign emergency stations (fire stations) in a city.
- Allocate sale branch offices for a company.
- Schedule for bus drivers.

A greedy approximation algorithm

Greedy-Set-Cover(X, F)

1. U = X
2. C = $\varnothing$
3. **while** U != $\varnothing$ **do**
4. select an $S \in F$ that maximizes $|S \cap U|$
5. U = U – S
6. $C = C \cup \{S\}$
7. **return** C

The algorithm GREEDY-SET-COVER can easily be implemented to run in time complexity in |X| and |F|. Since the number of iterations of the loop on line 3-6 is at most min(|X|, |F|) and the loop body can be implemented to run in time O(|X|,|F|), there is an implementation that runs in time O(|X|,|F| min(|X|,|F|) .

Ratio bound of Greedy-set-cover

- Let denote the *d*th harmonic number

  $$h_d = \sum_{i-1}^{d} 1/i$$

- Theorem: Greedy-set-cover has a ratio bound H(max{|S|: S $\in$ F})
- Corollary: Greedy-set-cover has a ratio bound of (ln|X| +1)

  (Refer to the text book for the proofs)

3. The Traveling Salesman Problem

- Since finding the shortest tour for TSP requires so much computation, we may consider to find a tour that is almost as short as the shortest. That is, it may be possible to find *near-optimal* solution.

- Example: We can use an approximation algorithm for the HCP. It's relatively easy to find a tour that is longer by at most a factor of *two* than the optimal tour. The method is based on
  - the algorithm for finding the *minimum spanning tree* and
  - an observation that it is always cheapest to go directly from a vertex *u* to a vertex *w*; going by way of any intermediate stop *v* can’t be less expensive.

$$C(u,w) \leq C(u,v) + C(v,w)$$

APPROX-TSP-TOUR

- The algorithm computes a near-optimal tour of an undirected graph G.

procedure APPROX-TSP-TOUR(G, c);

begin

select a vertex r ∈ V[G] to be the "root" vertex;

grow a minimum spanning tree T for G from root r, using Prim's algorithm;

apply a preorder tree walk of T and let L be the list of vertices visited in the walk;

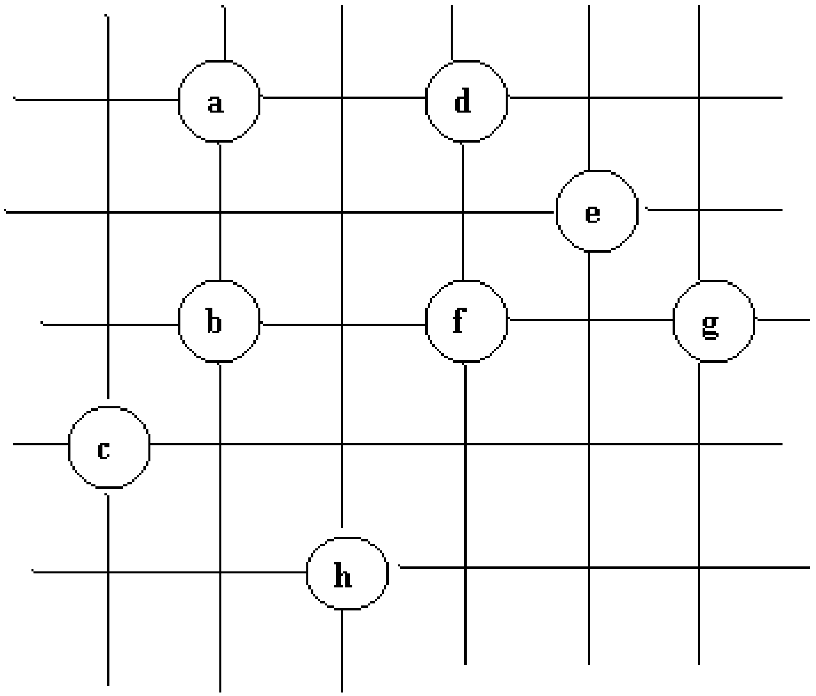
form the halmintonian cycle H that visits the vertices in the order of L.

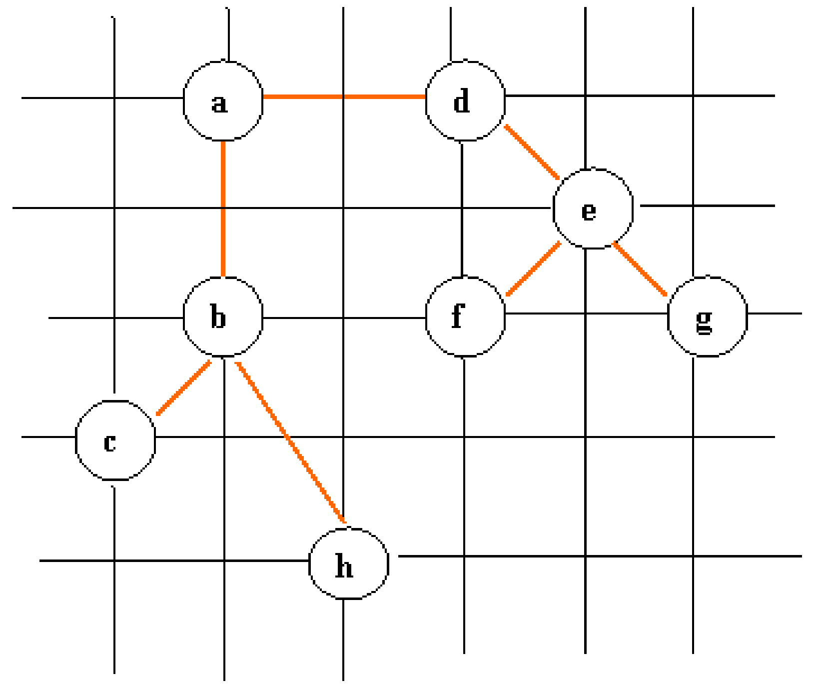
/* H is the result to return * /

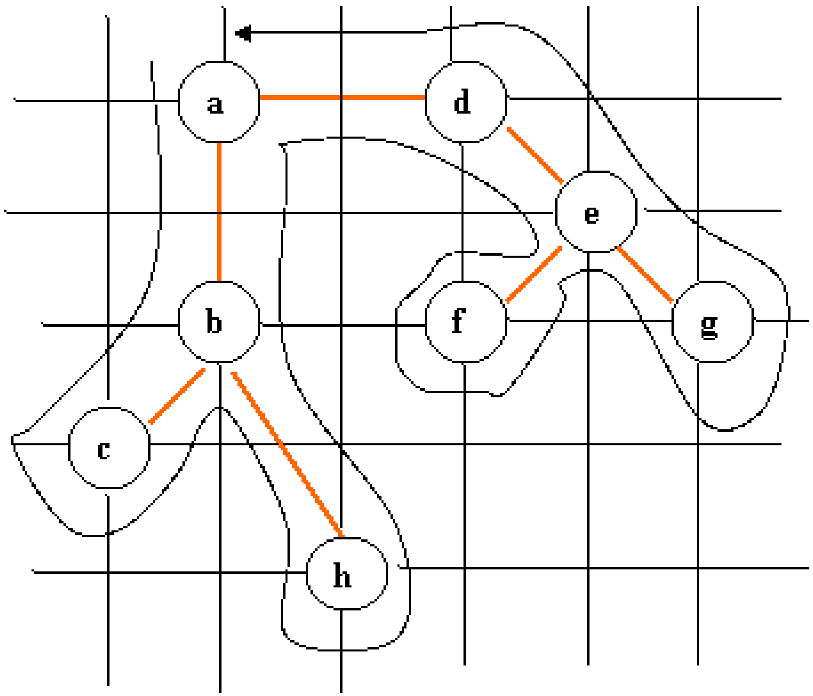
end

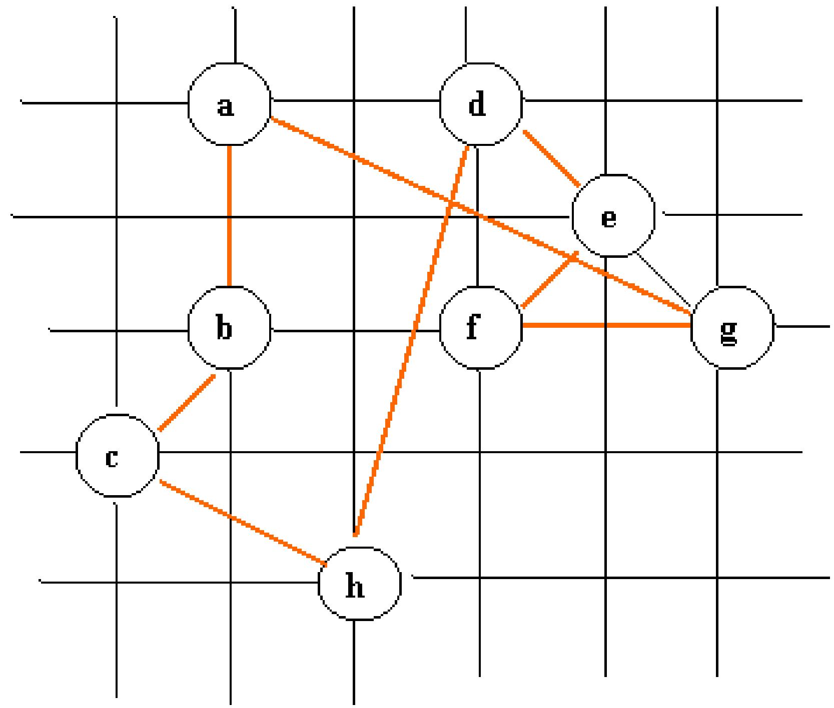
A *preorder tree walk* recursively visits every vertex in the tree, listing a vertex when its first encountered, before any of its children are visited.

Thí dụ minh họa giải thuật APPROX-TSP-TOUR

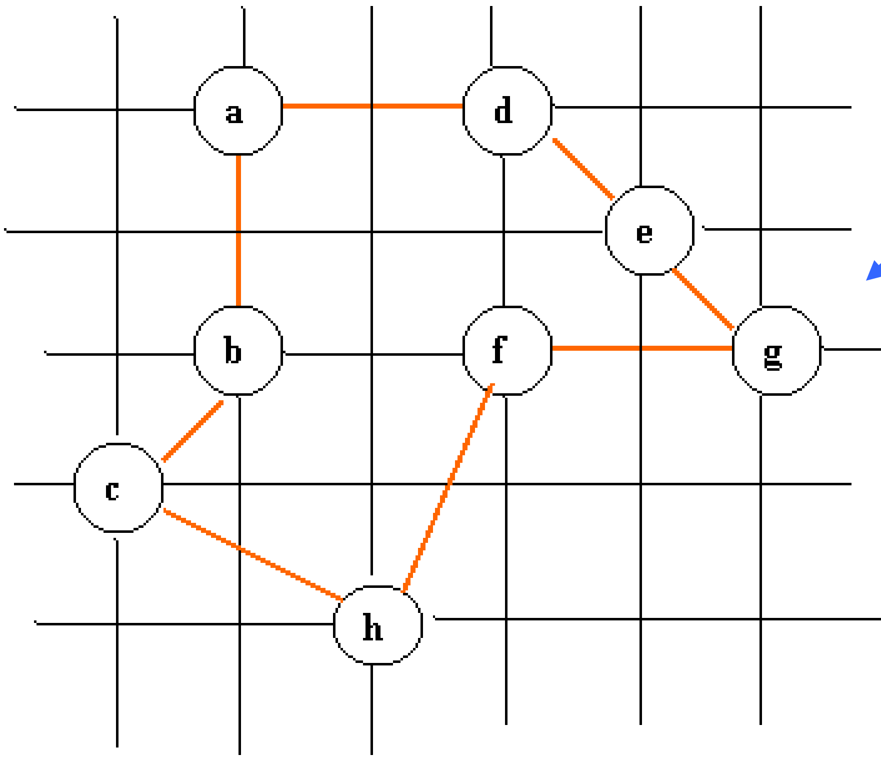








The preorder tree walk is not simple tour, since a node be visited many times, but it can be fixed, the tree walk visits the vertices in the order *a, b, c, b, h, b, a, d, e, f, e, g, e, d, a.* From this order, we can arrive to the hamiltonian cycle H: *a, b, c, h, d, e ,f, g, a.*



The optimal tour

The total cost of H is approximately 19.074. An optimal tour H* has the total cost of approximately 14.715.

The running time of APPROX-TSP-TOUR is $O(E) = O(V^2)$, since the input graph is a complete graph.

Ratio bound of APPROX-TSP-TOUR

- Theorem: APPROX-TSP-TOUR is an approximation algorithm with a ratio bound of 2 for the TSP with triangle inequality.
- Proof: Let H* be an optimal tour for a given set of vertices. Since we obtain a spanning tree by deleting any edge from a tour, if T is a minimum spanning tree for the given set of vertices, then

  $$c(T) \leq c(H^*) \qquad (1)$$

- A full walk of T traverses every edge of T twice, we have:

  $$c(W) = 2c(T) \qquad (2)$$

  (1) and (2) imply that:

  $$c(W) \leq 2c(H^*) \qquad (3)$$

- But *W* is not a tour, since it visits some vertices more than once. By the triangle inequality, we can delete a visit to any vertex from *W*. By repeatedly applying this operation, we can remove from *W* all but the first visit to each vertex.
- Let *H* be the cycle corresponding to this preorder walk. It is a hamiltonian cycle, since every vertex is visited exactly once. Since *H* is obtained by deleting vertices from W, we have

  $$c(H) \le c(W) \qquad (4)$$

- From (3) and (4), we conclude:

  $$c(H) \le 2c(H^*)$$

So, APPROX-TSP-TOUR returns a tour whose cost is not more than twice the cost of an optimal tour.

Appendix: A Taxonomy of Algorithm Design Strategies

Strategy name	**Examples**
Bruce-force	Sequential search, selection sort
Divide-and-conquer	Quicksort, mergesort, binary search
Decrease-and-conquer	Insertion sort, DFS, BFS
Transform-and-conquer	heapsort, Gauss elimination
Greedy	Prim's, Dijkstra's
Dynamic Programming	Floyd's
Backtracking	
Branch-and-Bound	
Approximate algorithms	
Heuristics	
Meta-heuristics	