Hamiltonian Cycles and Traveling Salesman Problem (TSP)

For the flight schedule given in Fig. 12.14 (p. 625 of GTM), is there a tour that takes us to each city exactly once and then takes us back to the starting city?

Such a tour is called a Hamiltonian Cycle, and a graph containing a Hamiltonian cycle is called Hamiltonian

Given a weighted graph, the Traveling Salesman Problem asks to find a Hamiltonian cycle with minimal weight (or of weight less than a given bound)
Example

Fig. 10.2.10 lists some cities in India with the approximate road distances in miles. Is there a tour that would take us to all of the cities in less than 4500 miles?

If we avoid the long route from Chennai to Kolkata, we can try: Chennai-Nagpur-Kolkata-Agra-Delhi-Jaiselmer-Jaipur-Mumbai-Chennai for a total distance of 4525 miles

Or perhaps we should try to avoid the expensive routes to Nagpur from Chennai and Kolkata: Chennai-Kolkata-Agra-Nagpur-Jaipur-Delhi-Jaiselmer-Mumbai-Chennai for a total distance of 4905 miles

Chennai-Kolkata-Nagpur-Agra-Jaipur-Delhi-Jaiselmer-Mumbai-Chennai is only 4405 miles: tour found!

How do we find the Hamiltonian cycle?
Finding Hamiltonian Cycle by Randomized Method

class Tour {
    int n; // number of cities
    int c[MAXCITIES];
    int l; // tour length
};

Find a Hamiltonian cycle in G and sum up its weighted path length

What is the worst-case running time of the algorithm?

Tour
hamiltonian(G)
{
    found = false; tour.n = n = |V|;
    while (!found) {
        for (i = 0 to n-1) {
            tour.c[i] = random_pick(V);
            V -= tour.c[i];
        }
        if ((c[n-1],c[0]) in E) {
            tour.l = weight(c[n-1],c[0]);
            found = true;
        } else { found = false; }
        for (i = 0 to n-1 && found) {
            if ((c[i],c[i+1]) in E) {
                tour.l += weight(c[i],c[i+1]);
            } else { found = false; }
        }
    }
    return tour;
}
Finding TSP by Brute-Force (BF) Method

Tour
tsp()
{
    best_tour = hamiltonian(G);
tsp = { 1, [1], 0 };
    BFTsp(&tsp, &best_tour);
    return best_tour;
}

Assume non-existent edges have weight $\infty$

What is the running time of this algorithm?

BFTsp(Tour *tsp, *best)
{
    n = tsp->n;
    if (n < best->n) {
        for all j neighbor of tsp->c[n-1]
in (best->c - tsp->c) {
            ntsp = *tsp;
            ntsp.l += weight(ntsp.c[n-1], j);
            ntsp.c[n] = j;
            ntsp.n++;
            BFTsp(&ntsp, best);
        }
    } else {
        newl = tsp->l +
            weight(tsp->c[n-1], tsp->c[0]);
        if (newl < best->l) *best = *tsp;
    }
}
Problem Space vs. Solution Space

While you’re searching for a solution in the problem space, you are generating “states” in the solution space.

The “states” in the solution space form a search tree, with each internal node being a partial solution and the leaves potential solutions.

For example:

- problem space: find a tsp tour starting from Chennai
- solution space: each step you take forms a partial path that is a unique “state” in the solution search tree.
Solution Search

When we talk of searching for a solution, we’re talking about searching the solution space/search tree

Methods/design patterns:

- brute force
- greedy, usually involving heuristics
- divide-and-conquer, usually involving recursion
- amortized
- randomized
- branch-and-bound and backtracking
- approximation
- local search
- dynamic programming
- steepest ascent hill climbing
- simulated annealing
- taboo search
- etc.
Brute Force

When is Brute Force useful?

- no smarter algorithms are in sight (they may or may not exist)
- to evaluate heuristics (e.g., greedy)
- when smart algorithm are very complicated and results need to be verified
  - implement a simple brute-force algorithm
  - see if smarter alg. produces correct solution

Often relies on **enumeration**: let’s try to produce all possible permutations of $N$ items (e.g., cities)

Takes $O(N \cdot N!)$ (tsp) or $O(N \cdot 2^N)$ (coin change)
Branch-and-Bound and Backtracking

**Branch**: enumerate all possible next steps from current partial solution

**Bound**: if a partial solution violates some constraint, e.g., an upper bound on cost, drop/prune the branch (don’t follow it further)

**Backtracking**: once a branched is pruned, move back to the previous partial solution and try another branch
Finding TSP by Naive Branch-and-Bound (NBnB) Method

Tour
tsp()
{
    best_tour = hamiltonian(G);
    tsp = { 1, [1], 0 };
    NBnBtsp(&tsp, &best_tour);
    return best_tour;
}

Assume non-existent edges have weight $\infty$

What is the worst-case running time of this algorithm?

NBnBtsp(Tour *tsp, *best)
{
    n = tsp->n;
    if (n < best->n) {
        for all j neighbor of tsp->c[n-1]
            in (best->c - tsp->c) {
                ntsp = *tsp;
                ntsp.l += weight(ntsps.c[n-1], j);
                if (ntsp.l < best->l) {
                    ntsps.c[n] = j;
                    ntsps->n++;
                    NBnBtsp(&ntps, best);
                } // otherwise bound and backtrack
            }
    } else {
        newl = tsp->l +
            weight(tsp->c[n-1], tsp->c[0]);
        if (newl < best->l) *best = *tsp;
    }
}
BnB Efficiency

Two factors affect the efficiency of BnB:

1. how soon (high up in the solution search tree) you can prune away a partial solution: the sooner you can make the call to prune, the less time you need to spend on the branch

2. for optimization problem (e.g., finding the min cost tour) the tightness of your initial bound could be a huge factor in performance; so it may be worth spending extra effort to compute better bounds

This is where you will spend the bulk of your time in PA3

Assumed implementation of any given algorithm/heuristics has been optimized from a systems perspective (e.g., there will be no point given implementing adjacency list instead of adjacency matrix)
PA3 Notes

Allowed to search the Internet for potential solutions. However:

- Do not consult previous terms’ solutions to this programming assignment
- **MUST** cite sources (or solution will **not** be graded)
- Don’t go off tangent reading more and more minute incremental improvements
- Improvements outside BnB won’t be accepted, in particular, the following improvements will not be accepted:
  - random walks
  - simulated annealing
  - taboo search
  - Lin-Kernighan
  - dynamic programming

- Why? So that you won’t go off tangent spending too much time. Save them for EECS 477
Solution Search

When we talk of searching for a solution, we’re talking about searching the solution space/search tree

Methods/design patterns:

- brute force
- greedy, usually involving heuristics
- divide-and-conquer, usually involving recursion
- amortized
- randomized
- branch-and-bound and backtracking
- approximation
- local search
- dynamic programming
- steepest ascent hill climbing
- simulated annealing
- taboo search
- etc.
A 2-Approximation for a Special Case of TSP

If distances satisfy the **triangle inequality**, i.e., for edges \((u, v), (v, w), (u, w)\) in \(G : C(u, v) + C(v, w) \geq C(u, w)\), and \(G\) is a **complete graph**, we can build a 2-approximate TSP in three steps:

1. construct an MST of the graph (using Prim’s for example)
2. construct an Euler-cycle of the MST
3. replace all edges \((u, v)\) and \((v, w)\) for which \(v\) is visited more than once with edge \((u, w)\)

\(k\)-approximate means that the result is off by a factor of \(k\) from the optimal

Running time complexity:
Euler Cycle

An **Euler cycle** in a graph $G$ is a path from $v$ to $v$ with no repeated edges that contains all of the edges and all of the vertices of $G$.

The Könisberg-bridge problem: the first graph theoretic problem!

A graph $G$ has an Euler cycle iff $G$ is connected and the degree of every vertex is even.

In our case, each MST edge is considered as 2 directed edges in separate direction, so each vertex also has an even number of edges.
2-Approximate Proof

Let:

- $M$ be the MST
- $E$ the Euler tour
- $T$ the non-optimal TSP obtained from $E$
- and $T'$ the optimal TSP minus an edge, which must be a spanning tree

Then:

- $C(M) \leq C(T')$ by definition of MST
- $C(E) = 2C(M)$
- $C(T) \leq C(E)$ by triangle inequality

So $C(T) \leq 2C(T')$
**k-Opt**

**k-Opt** is a greedy local search heuristic:

- start with an initial tour, e.g., the randomly generated one or the MST
- consider two cities \( v_i \) and \( v_{i+1} \) or any two cities \( u \) and \( v \)
- check if visiting \( v_{i+1} \) first before \( v_i \) decreases total tour length
- repeat with different \( v_i \) until there is no further improvement (or up to time limit)

The above is 2-opt, you can also consider other \( k \) values.
Computing the Fibonacci Sequence

What is a Fibonacci sequence?

How would you implement it?
Computing the Fibonacci Sequence (contd)

What is a Fibonacci sequence?
\[ f_0 = 0; f_1 = 1; f_n = f_{n-1} + f_{n+2}, n \geq 2 \]

Recursive implementation:

```c
int rfib(int n)
{ // assume n >= 0
    return (n <= 1 ? n : rfib(n-1)+rfib(n-2));
}
```

Running time: \( \Omega((\frac{3}{2})^n) \)

Iterative version:

```c
int ifib(int n)
{ // assume n >= 2
    int i, f[n];
    f[0] = 0; f[1] = 1;
    for (i = 2 to n) {
        f[i] = f[i-1]+f[i-2];
    }
    return f[n];
}
```

Running time: \( \Theta(n) \)
Computing the Fibonacci Sequence (contd)

Why is the recursive version so slow?

Why is the iterative version so fast?
Computing the Fibonacci Sequence (contd)

Why is the recursive version so slow?
The number of computations grows exponentially!
Each \texttt{rfib}(i), i < n - 1 computed more than once.
Tree size grows almost \(2^n\). Actually the number of base case computations in computing \(f_n\) is \(f_n\).
Since \(f_n > \left(\frac{3}{2}\right)^{n-1}\), complexity is \(\Omega\left((\frac{3}{2})^n\right)\).

Why is the iterative version so fast?
Instead of recomputing duplicated subproblems, it saves their results in an array and simply looks them up as needed.

Can we design a recursive algorithm that similarly look up results of duplicated subproblem?
Memoized Fibonacci Computation

```c
int fib_memo[n] = {0, 1, -1, . . . , -1};
int
mfib(int n, *fib_memo)
{ // assume n >= 0 and left to right evaluation
    if (fib_memo[n] < 0)
        fib_memo[n] = mfib(n-2, fib_memo) + mfib(n-1, fib_memo);
    return fib_memo[n];
}
```

Memoization (or tabulation): use a result table with an otherwise inefficient recursive algorithm. Record in the table values that have been previously computed.

Memoize only the last two terms:

```c
int
rfib2(int fn2, fn1, n)
{ // assume n >= 0
    return (n <= 1 ? n:
        rfib2(fn1, fn2+fn1, n);
    }
main() { return rfib2(0, 1, n); }
```
Divide et impera

Divide-and-conquer:

- for base case(s), solve problem directly
- do recursively until base case(s) reached:
  - divide problem into 2 or more subproblems
  - solve each subproblem independently
- solutions to subproblems combined into a solution to the original problem

Works fine when subproblems are non-overlapping!
Otherwise overlapping subproblems must be solved more than once.

Sugih Jamin (jamin@eecs.umich.edu)
Dynamic Programming

- used when a problem can be divided into subproblems that overlap
- solve each subproblem once and store the solution in a table
- if run across the subproblem again, simply look up its solution in the table
- reconstruct the solution to the original problem from the solutions to the subproblems
- the more overlap the better as this reduces the number of subproblems

Origin of name:
**Programming**: planning, decision making by a tabular method
**Dynamic**: multi-stage, time-varying process
Dynamic Programming and Optimization Problem

DP used primarily to solve optimization problem, e.g., find the shortest, longest, “best” way of doing something.

Requirement: an optimal solution to the problem must be a composition of optimal solutions to all subproblems. There must not be an optimal solution that contains suboptimal solution to a subproblem.
DP for TSP

Observation:

Let the optimal (shortest) TSP path from node $v_1$ be $[v_1, v_k, v_{k+1}, v_{k+2}, \ldots, v_{k+n}, v_1]$.

The subpath $[v_k, v_{k+1}, v_{k+2}, \ldots, v_{k+n}, v_1]$ must be the shortest path from $v_k$ to $v_1$ that visits each of the other vertices exactly once.

Similarly for $[v_{k+1}, v_{k+2}, \ldots, v_{k+n}, v_1]$, $[v_{k+2}, \ldots, v_{k+n}, v_1]$, etc., down to $[v_{k+n}, v_1]$.

So, what’s the problem?
DP for TSP

Problem is we don’t know which permutation of \(\{v_k, v_{k+1}, v_{k+2}, \ldots, v_{k+n}\}\) gives the shortest path.

We have to try them all out:

\[
\begin{align*}
\{v_k, v_{k+1}, v_{k+2}, \ldots, v_{k+n-2}, v_{k+n-1}, v_{k+n}\} \\
\{v_k, v_{k+1}, v_{k+2}, \ldots, v_{k+n-1}, v_{k+n-2}, v_{k+n}\} \\
\ldots \\
\{v_k, v_{k+2}, v_{k+1}, \ldots, v_{k+n-2}, v_{k+n-1}, v_{k+n}\} \\
\{v_{k+1}, v_k, v_{k+2}, \ldots, v_{k+n-2}, v_{k+n-1}, v_{k+n}\} \\
\{v_{k+2}, v_k, v_{k+1}, \ldots, v_{k+n-2}, v_{k+n-1}, v_{k+n}\} \\
\ldots \\
\{v_k, v_{k+1}, v_{k+2}, \ldots, v_{k+n-2}, v_{k+n}, v_{k+n-1}\} \\
\{v_k, v_{k+1}, v_{k+2}, \ldots, v_{k+n}, v_{k+n-2}, v_{k+n-1}\} \\
\ldots
\end{align*}
\]

and all other permutations thereof.

Note the amount of duplicated subproblems!
Example: Computing TSP using DP

Given the following graph:

Store the edge weights in an adjacency matrix with $\infty$ as the weight of a non-existent edge:

\[
\begin{array}{cccc}
  & A & B & C & D \\
A & 0 & 2 & 9 & \infty \\
B & 1 & 0 & 6 & 4 \\
C & \infty & 7 & 0 & 8 \\
D & 6 & 3 & \infty & 0 \\
\end{array}
\]

Let $w()$ the weight of an edge and $d([])$ be the length of a path
Computing TSP using DP

**Last hop:**
\[d([B, A]) = w(B, A) = 1\]
\[d([C, A]) = w(C, A) = \infty\]
\[d([D, A]) = w(D, A) = 6\]

**Second to last hop:**
\[d([B, C, A]) = w(B, C) + d([C, A]) = 6 + \infty = \infty\]
\[d([B, D, A]) = w(B, D) + d([D, A]) = 4 + 6 = 10\]
\[d([B, \ldots, A]) = MIN (d([B, C, A]), d([B, D, A])) = 10\]
\[d([C, B, A]) = w(C, B) + d([B, A]) = 7 + 1 = 8\]
\[d([C, D, A]) = w(C, D) + d([D, A]) = 8 + 6 = 14\]
\[d([C, \ldots, A]) = MIN (d([C, B, A]), d([C, D, A])) = 8\]
\[d([D, B, A]) = w(D, B) + d([B, A]) = 3 + 1 = 4\]
\[d([D, C, A]) = w(D, C) + d([C, A]) = \infty + \infty = \infty\]
\[d([D, \ldots, A]) = MIN (d([D, B, A]), d([D, C, A])) = 4\]
Example: DP for TSP (contd)

Third to last hop:

\[ d([B, \ldots, A]) = \min(\sum_{i=0}^{n} w(B, D) + d([D, C, A]), \sum_{i=0}^{n} w(B, C) + d([C, D, A])) \]
\[ = \min(4 + \infty, 6 + 14) \]
\[ = 20 \]

\[ d([C, \ldots, A]) = \min(\sum_{i=0}^{n} w(C, D) + d([D, B, A]), \sum_{i=0}^{n} w(C, B) + d([B, D, A])) \]
\[ = \min(8 + 4, 7 + 10) \]
\[ = 12 \]

\[ d([D, \ldots, A]) = \min(\sum_{i=0}^{n} w(D, B) + d([B, C, A]), \sum_{i=0}^{n} w(D, C) + d([C, B, A])) \]
\[ = \min(3 + \infty, \infty + 8) \]
\[ = \infty \]
Example: DP for TSP (contd)

Full path:

\[ d([A, \ldots, A]) = \min(\ w(A, B) + d([B, \ldots, A]), \]
\[ w(A, C) + d([C, \ldots, A]), \]
\[ w(A, D) + d([D, \ldots, A])) \]
\[ = \min(2 + 20, 9 + 12, \infty + \infty) \]
\[ = 21 \]

Extracted path (TSP):

\[ [A, [C, \ldots, A]] = [A, [C, [D, \ldots, A]]] = [A, [C, [D, [B, A]]]] \]
Complexity of DP for TSP

Time complexity: $\Theta(n^22^n)$
Space complexity: $\Theta(n2^n)$

Still exponential, not polynomial! What good is that?
Compared against the brute force algorithm with time complexity $O(n!)$, this is a big improvement

Assuming 1 $\mu$sec per “basic computation” in each algorithm, to compute a TSP of 20 cities:

- brute force takes 3,857 years
- dp takes 45 seconds, using about 40 MB of memory
- for 60 cities, dp will also take many years . . . .