COMP 362 Assignment 1

Christopher Hundt 110220945

January 27, 2004

1. We first note that a specific combination T of a set S of n cities can be represented by an n-bit number $b_n b_{n-1} \cdots b_2 b_1$ where $b_i = 1 \iff c_i \in T$ for a given numbering of the cities $c_1, \ldots, c_n$.

   Thus, the algorithm uses one array T of size 2^n, with each element of the array pointing to one hash h_k. The indexing of the array starts at 1, so that any combination with at least one city corresponds to exactly one index in the array using the above binary representation. Then the array element points to a hash. The hash has some subset of the integers from 1 to n as indices, and each index i contains in its key the optimal cost C for a tour with c_i as the last city and with the tour including the subset of cities indicated by the array index which pointed to the hash. In fact the key is an ordered pair (C, j) with j being the index of the previous city in an optimal tour.

   As an example, suppose we are solving a ten-city salesman problem with cities $c_1, \ldots, c_{10}$, and the current subproblem is all of the cities except for c_1 and c_3, with c_5 being the last city. The algorithm finds an optimal tour with cost 83 and with c_8 being the second-to-last city in the tour. Then the index involved is $1111111010_{\text{two}} = 1018$. Then $t[1018]$ points to h_{1018}, so we set $h_{1018}[5] \leftarrow (83, 8)$.

   The algorithm, which evaluates the function *TSP* given in class, is as follows:

```
TRAVELING-SALESMAN(S, l)
 1  if T[s] has key l
 2     then return T[S][l]
 3     else  if S is an integral power of 2 ("S has one digit")
 4              then T[S][l] ← (C_{1l}, 1)
 5                   return (C[1][l], 1)
 6           (M_c, M_l) ← (∞, NIL)
 7           S' ← S with l-th digit = 0 ("S \ c_l")
 8           for i ← 1 to n
 9                do if i-th digit of S' is 1 ("c_i ∈ S'")
10                      then (r_c, r_l) ← TRAVELING-SALESMAN(S', i)
11                           t ← r_c + C_{il}
12                           if t < M_c
13                              then (M_c, M_l) ← (t, i)
14           T[S][l] ← (M_c, M_l)
15           return (M_c, M_l)
```

   Note that it is easy to get "S with the l-th digit removed" on a computer in constant time using simple binary operations. It is similarly simple to see if the i-th digit is equal to 1 in constant time. There is also a variety of easy numerical ways to check if a number S is an integral power of 2.

   Now that we can get the optimal cost, we use a second algorithm and the already-computed information to find an actual minimal cost tour for cities $c_1, \ldots, c_n$:

TRAVELING-SALESMAN-TOUR

```
1   S ← 11···1 (n digits)_two = 2^n − 1
2   (r_c, r_l) ← TRAVELING-SALESMAN(S, 1)
3   t ← (c_1)
4   k ← r_l
5   S ← S with 1st digit = 0 ("S \ c_1")
6   while S ≠ 0
7         do add c_k to the beginning of sequence t
8            (r_c, r_l) ← T[S][k]
9            S ← S with k-th digit = 0 ("S \ c_k")
10           k ← r_l
11  add c_1 to the beginning of t
12  return t
```

This algorithm computes the optimal tour, then uses the data in the array C and its associated hashes to recreate the tour in $\Theta(n)$ time. Because $T[S][l]$ contains not only the cost but also the second to last stop k on the tour of S ending with l, you can take $T[S \setminus l][k]$ to find the stop before k, and so on until you have recreated the entire tour.

2. In order for a parenthesization to be unambiguous, there must be $n-1$ sets of parentheses for n matrices. This is because to be unambiguous each pair of parentheses must contain exactly two "objects" in it, where an object is either a matrix or another set of parentheses. Thus three matrices requires one pair of parentheses, and each time you add a matrix after that you need to add one more pair of parentheses.

   Now we observe that, by parenthesizing, we split an expression into two smaller expressions. We could have $A_1(A_2 \cdots A_n)$, or $(A_1A_2)(A_3 \cdots A_n)$, or $(A_1A_2 \cdots A_{n-1})A_n$, or anything in between. That is, we split it into two parenthesized expressions, one with k matrices and one with $n-k$ matrices. Thus, if the total number of possibile parenthesizations is P_n, the number of parenthesizations where the outermost splitting occurs after the k-th matrix is P_kP_{n-k}. Thus the total number P_n is

$$P_n = \sum_{k=1}^{n-1} P_k P_{n-k},$$

   with $P_1 = P_2 = 1$. This becomes the Catalan sequence, where $P_n = \frac{1}{n+1}\binom{2n}{n}$. Using Stirling's formula,

$$\begin{aligned} P_n &= \frac{1}{n+1}\binom{2n}{n} &&= \frac{1}{n+1}\cdot\frac{(2n)!}{n!n!} \\ &&&\sim \frac{\sqrt{2\pi}\,(2n)^{\left(2n+\frac{1}{2}\right)}\,e^{-2n}}{(n+1)\left(\sqrt{2\pi}n^{\left(n+\frac{1}{2}\right)}e^{-n}\right)^2} \\ &&&= \frac{(2n)^{2n}}{(n+1)\,\sqrt{\pi}n^{\left(2n+\frac{1}{2}\right)}} \\ &&&= \frac{4^n}{\sqrt{\pi}\left(n^{\frac{3}{2}}+\sqrt{n}\right)}, \end{aligned}$$

   so then

$$P(n) = \frac{4^n}{\sqrt{\pi}n^{\frac{3}{2}}}\left(1+o(1)\right).$$

3. We first note that, if the paritioning into $a_1 \dots a_{i_1} | a_{i_1+1} \dots a_{i_2} | \dots | a_{i_{m-2}+1} \dots a_{i_{m-1}} | a_{i_{m-1}+1} \dots a_n$ is minimal, then the partitioning of $a_{i_1+1} \dots a_n$ into $m-1$ partitions must be optimal. To prove this, assume it is not. Then there is some new partitioning of $a_{i_1+1} \dots a_n$ into $m-1$ partitions with minimal cost, which means that adding back the partition $a_1 \dots a_{i_1}$ will get a partitioning of $a_1 \dots a_n$ into m parts with less cost then the original partition of minimal cost, a contradiction.

   Thus, we can find the minimum cost with the following recursive function defined for the sequence $A = (a_1, a_2, \dots, a_n)$ and number of partitions m:

$$P(A,m) = \begin{cases} \left(\sum_{k=1}^{n} s_k\right)^3, & \text{if } m = 1; \\ \min_{1 \le i \le n-m+1} \left(\left(\sum_{k=1}^{i} s_k\right)^3 + P(\{a_{i+1}, a_{i+2}, \dots, a_n\}, m-1) \right), & \text{otherwise.} \end{cases}$$

   Thus we simply need to write a recursive algorithm that implements that function but uses an array to save previous values so they are not recalculated. The algorithm also needs to save the i chosen in the second part of the function definition so that the actual partitioning can be recreated.

   So we define an $m \times (n-m+1)$ array P (with indexing starting at 1) such that, when the algorithm is done, $P_{ij} = (c, x)$, where c is the minimum cost for dividing $a_i, \dots, a_n$ into m partitions, and a_x is where the outermost parenthetical splitting occurs. Here is the algorithm, which assumes that it has access to the length n of the original sequence A:

```
PARTITION-COST(A, m)
 1  j ← n − |A| + 1
 2  if P_mj is defined
 3      then return P_mj
 4      else if m = 1
 5              then S ← 0
 6                   for k ← 1 to |A|
 7                        do S ← S + a_k
 8                   P_mj ← (S^3, NIL)
 9                   return (S^3, NIL)
10              else (M_c, M_x) ← (∞, NIL)
11                   for i ← 1 to |A| − m + 1
12                        do S ← 0
13                           for k ← 1 to i
14                                do S ← a_i
15                           S ← S^3 + PARTITION-COST({a_{i+1}, ..., a_{|A|}}, m − 1)
16                           if S < M_c
17                              then (M_c, M_x) → (S, i + n − |A|)
18                   P_mj ← (M_c, M_x)
19                   return (M_c, M_x)
```

   Now, to get the actual partitioning, we use the following algorithm:

```
PARTITIONING(A, m)
1  (r_c, r_x) ← PARTITION-COST(A, m)
2  X ← (a_{r_x})
3  while r_x ≠ NIL
4        do m → m − 1
5           (r_c, r_x) ← PARTITION-COST(A \ {r_x}, m)
6           add a_{r_x} to the beginning of sequence X
7  return X
```

This algorithm recreates the optimal partitioning and returns a sequence of the $m-1$ elements of A that are the last elements of each of the first $m-1$ partitions.

We can measure the total time for PARTITION-COST in the following way: We know that it will be run once for each possible combination of values of m and n. Thus it will be run mn times. Although there are recursive calls, they take constant time after the first time thanks to the array P, so we can simply count all recursive calls as constant time because we are already counting going through the algorithm once for each combination of m and n.

Suppose we have an original sequence of size n and want m partitions. Now, when running PARTITION-COST(A, m'), we see that the loop on line 6 runs $|A|$ times. The loop on line 11 runs $|A|-m'+1$ times, and the nested loop on line 13 iterates, on average, about $\frac{1}{2}(|A|-m'+1)$ times for each iteration of line 11's loop. Thus the total number of iterations of line 13's loop is $O((|A|-m')^2)$ and the loop of line 6 is insignificant. Now m' can be anything from 1 to m, and then $|A|$ can be anything from m' to n (note that not all the entries in the array are filled because you can't have a sequence with x elements and y partitions where $x < y$). So then the total number N of iterations of line 13's loop over all possible calls to PARTITION-COST is

$$\begin{aligned}
N &\leq \sum_{m'=1}^{m}\sum_{k=m}^{n}(k-m)^2 \\
&= \sum_{m'=1}^{m}\sum_{k=m}^{n}\left(k^2+m^2-2km\right) \\
&< \sum_{m'=1}^{m}\sum_{k=m}^{n}\left(k^2+m^2\right) \\
&= \sum_{m'=1}^{m}\left((n-m)\,m^2+m^2+(m+1)^2+\ldots+n^2\right) \\
&< \sum_{m'=1}^{m}\left((n-m)\,m^2+(n-m)\,n^2\right) \\
&< \sum_{m'=1}^{m}2n^3 \;=\; 2mn^3 \in O(mn^3).
\end{aligned}$$

The time for PARTITIONING after the initial call to PARTITION-COST is $\Theta(m)$, since all the later calls to PARTITION-COST take time $\Theta(1)$, so the total time is at worst polynomial in m and n, $O(mn^3)$.

The space required for the algorithm is even easier to calculate. The array has fewer than mn entries, and each entry is a pair, where the maximum value of each item in the pair is n for the second item, and the cost of the partitioning as the second entry. Assuming that the value of an individual item in A is bounded by C, then the cost can be represented in at most $\log_2(nC)^3 \in O(\log nC)$. So the total space cost is $O(mn+\log nC)$.

4. We first note that the pair (S, I) where I is the set of subsets of S that are matchable into T, is a matroid.

   - **hereditary:** Consider a set $A \in I$ and a subset $B \subseteq A$. Then any subset $C \subseteq B$ is also a subset of A, so it will have the property that $|C| \leq |N(C)|$.
   - **exchange:** Suppose we have $A, B \in I$ where $|A| < |B|$. Then either $|N(A)| \geq |N(B)|$ or $|N(A)| < |N(B)|$. Suppose $|N(A)| \geq |N(B)|$. Then take any element $x \in B \setminus A$ and $|A \cup \{x\}| \leq |B| \leq |N(B)| \leq |N(A)|$. Otherwise, suppose $|N(A)| < |N(B)|$. Then there is some $y \in T$ such that $(b, y) \in R$ for some $b \in B$, but $(a, y) \notin R$ for all $a \in A$. Let b be such an element of B. Then $N(A \cup \{b\})$ has this element y, whereas $N(A)$ did not. So $|N(A \cup \{b\})| \geq |N(A)| + 1$. Thus, $|A \cup \{b\}| = |A| + 1 \leq |N(A)| + 1 \leq |N(A \cup \{b\})|$.

   Now we know we can use the greedy algorithm to find a matchable set of maximum weight. We can test for whether a set is matchable using the graph equivalency between matchable subsets and matching graphs.

```
MATCHABLE(S, T, w, R)
 1  D ← empty dictionary
 2  X ← ∅
 3  order the elements of S in non-increasing order of w(S)
 4  for each s ∈ S using above order
 5        do t ← ∅
 6           for each (x, y) ∈ R where x = s
 7                 do k ← TRUE
 8                    for each (x', y') ∈ X where y' = y
 9                          do if D[x'] contains more than just y
10                                then delete y from D[x']
11                                else  k ← FALSE
12                    if k = TRUE
13                       then t ← t ∪ {y}
14                            X ← X ∪ {(x, y)}
15           if t ≠ ∅
16              then D[s] ← t
17  return the set of keys of D
```

Basically, lines 5 through 16 of this algorithm decide whether a graph with some new edges remains a matching. Each vertex may have more than one edge leading out of it. So the algorithm adds all the edges (essentially, the dictionary D is the matching graph that is being formed), but remembers what vertices they are associated with. Then, when it wants to add a new vertex $v \in S$, it checks to see what other vertices the edges starting at v touch. If there is some edge e incident to v that touches some other vertex $v' \in T$ which is also incident to some other edge e' that has already been added to the matching graph, then the algorithm checks each other vertex $v'' \in S$ that e' is also incident to. If v'' is incident to any other edges in the matching graph, then e' is removed because v'' will still be connected. On the other hand, if v'' is only incident to e' then e is not added to the maximal matching subset of S, because the graph would no longer be matching (both v and v'' would have edges with v' as the other endpoint). If at least one edge incident to v can be added, then v is added with those edges at the dictionary entry indexed by v.

Using this method, the algorithm can fulfill the requirements for the basic greedy algorithm, so it is guaranteed to find an optimal solution.