Our solution will be similar to the algorithm for VERTEX COVER from the first section of this chapter. Consider the notation defined in the problem. For an element $a \in A$, we *reduce the instance by a* by deleting a from A, and deleting all sets B_i that contain a. Thus, reducing the instance by a producing a new, presumably smaller, instance of HITTING SET.

We observe the following fact. Let $B_i = \{x_1, \ldots, x_c\} \subseteq A$ be any of the given sets in the HITTING SET instance. Then at least one of $x_1, \ldots, x_c$ must belong to any hitting set H. So by analogy with (2.3) from the notes, we have the following fact

- Let $B_i = \{x_1, \ldots, x_c\}$ There is k-element hitting set for the original instance if and only if, for some $i = 1, \ldots, c$, the instance reduced by x_i has a $(k-1)$-element hitting set.

The proof is completely analogous to that of (2.3). If H is a k-element hitting set, then some $x_i \in H$, and so $H - \{x_i\}$ is a $(k-1)$-element hitting set for the instance reduced by x_i. Conversely, if the instance reduced by x_i has a $(k-1)$-element hitting set H', then $H' \cup \{x_i\}$ is a k-element hitting set for the original instance.

Thus, our algorithm is as follows. We pick any set $B_i = \{x_1, \ldots, x_c\}$. For each x_i, we recursively test if the instance reduced by x_i has a $(k-1)$-element hitting set. We return "yes" if and only if the answer to one of these recursive calls is "yes." Our running time satisfies $T(m,k) \leq cT(m,k-1) + O(cm)$, and so it satisfies $T(m,k) = O(c^k \cdot kcm)$. This gives the desired bound, with $f(c,k) = kc^{k+1}$ and $p(m) = m$.

[1] ex579.588.787

(a) We prove this by induction on d. If $d = 0$, then Φ is a satisfying assignment, and *Explore*(Φ, d) returns "yes."

Now consider $d > 0$. If *Explore*(Φ, d) returns "yes," it is because one of the recursive calls *Explore*$(\Phi_i, d-1)$ returns "yes"; by induction, this means that Φ_i has distance $d-1$ to a satisfying assignment, and so Φ has distance d to a satisfying assignment.

Conversely, suppose Φ has distance d to a satisfying assignment Φ'. Consider any clause unsatisfied by Φ; since Φ' satisfies it, it must disagree with Φ on the setting of at least one of the variables in this clause. Thus, one of the assignments Φ_i, which changes the assignment to this variable, is at distance $d-1$ to Φ'; by induction the recursive call *Explore*$(\Phi_i, d-1)$ will return "yes," and so the full call *Explore*(Φ, d) will also return "yes."

The running time for *Explore* satisfies the recurrence $T(n, d) \leq 3T(n, d-1) + p(n)$, for a polynomial p. Unwinding this to get d down to 0, we have a running time of $O(3^d \cdot p(n))$.

(b) We let Φ_0 denote the assignment in which all variables are set to 0, and we let Φ_1 denote the assignment in which all variables are set to 1. If there is any satisfying assignment, it is within distance at most $n/2$ of one of these, so we can call both *Explore*$(\Phi_0, n/2)$ and *Explore*$(\Phi_1, n/2)$, and see if either of these returns "yes."

The running time of each of these calls is $O(p(n) \cdot 3^{n/2}) = O(p(n) \cdot (\sqrt{3})^n)$.

[1] ex695.88.327

Consider an ordered triple (S, i, j), $1 \leq i, j \leq n$ and S is a subset of the vertices that includes v_i and v_j. Let $B[S, i, j]$ denote the answer to the question, "Is there a Hamiltonian path on $G[S]$ that starts at v_i and ends at v_j?" Clearly, we are looking for the answer to $B[V, 1, n]$.

We now show how to construct the answers to all $B[S, i, j]$, starting from the smallest sets and working up to larger ones, spending $O(n)$ time on each. Thus the total running time will be $O(2^n \cdot n^3)$.

$B[S, i, j]$ is true if and only if there is some vertex $v_k \in S - \{v_i\}$ so that (v_i, v_k) is an edge, and there is a Hamiltonian path from v_k to v_j in $G[S - \{v_i\}]$. Thus, we set $B[S, i, j]$ to be true if and only if there is some $v_k \in S - \{v_i\}$ for which $(v_i, v_k) \in E$ and $B[S - \{v_i\}, k, j]$ is true. This takes $O(n)$ time to determine.

[1] ex386.623.944

We claim that such a graph G has a tree decomposition $(T, \{V_t\})$ in which each piece V_t corresponds uniquely to an internal triangular face of G. We prove this by induction on the number of nodes in G.

Choose any internal edge $e = (u, v)$ of G; deleting u and v produces two components A and B. Let G_1 be the subgraph induced on $A \cup \{u, v\}$ and G_2 the subgraph induced on $B \cup \{u, v\}$. By induction, there are tree decompositions $(T_1, \{X_t\})$ and $(T_2, \{Y_t\}$ of G_1 and G_2 respectively in which the pieces correspond uniquely to internal faces. Thus there are nodes $t_1 \in T_1$ and $t_2 \in T_2$ that correspond to the faces containing the edge (u, v). If we let T denote the tree obtained by adding an edge (t_1, t_2) to $T_1 \cup T_2$, then $(T, \{X_t\} \cup \{Y_t\})$ is a tree decomposition having the desired properties.

[1] ex203.262.545

(a) Let us root the tree at an arbitrary node r, and define subtrees T_u as we have done in Chapter 10.2 when solving the Weighted Independent Set Problem. There we defined two subproblems corresponding to each subtree depending on whether or not we include the root u in the set. We will use the same subproblems: $OPT_{in}(u)$ denotes the maximum weight of an independent set of T_u that includes u, and $OPT_{out}(u)$ denotes the maximum weight of an independent set of T_u that does not include u. Now the optimum we are looking for is $\min(OPT_{in}(r), OPT_{out}(r))$. It helps us to define a third subproblem for each subtree: $OPT_{un}(u)$ denotes the maximum weight of an independent set of T_u that does not have to dominate u. When u's parent is included in the dominating set, u is already dominated by its parent, hence the set selected in the subtree T_u does not have to dominate u.

Now that we have our sub-problems, it is not hard to see how to compute these values recursively. For a leaf $u \neq r$ we have that $OPT_{out}(u) = \infty$, $OPT_{in}(u) = c(u)$, and $OPT_{un}(u) = 0$. For all other nodes u we get a recurrence that defines $OPT_{out}(u)$, $OPT_{in}(u)$, and $OPT_{un}(v)$ using the values for u's children.

(1) *For a node u, the following recurrence defines the values of the sub-problems:*

- $OPT_{in}(u) = c(u) + \sum_{v \in children(u)} OPT_{un}(v)$
- $OPT_{un}(u) = \sum_{v \in children(u)} \min(OPT_{in}(v), OPT_{out}(v))$
- $OPT_{out}(u) = \min_{v \in children(u)} (OPT_{in}(v) + \sum_{w \in children(u), w \neq v} \min(OPT_{out}(v), OPT_{in}(v)))$.

Using this recurrence, we get a dynamic programming algorithm by building up the optimal solutions over larger and larger sub-trees. We define arrays $Mo[u]$, $Mi[u]$ and $Mu[u]$, which hold the values $OPT_{out}(u)$, $OPT_{in}(u)$ and $OPT_{un}(u)$ respectively. For building up solutions, we need to process all the children of a node before we process the node itself.

To find a minimum-weight dominating set of a tree T:
  Root the tree at a node r.
  For all nodes u of T in post-order
    If u is a leaf then set the values:
      $Mo[u] = \infty$
      $Mi[u] = c(u)$
      $Mu[u] = 0$
    Else set the values:
      $Mi[u] = c(u) + \sum_{v \in children(u)} Mu[v]$.
      $Mu[u] = \sum_{v \in children(u)} \min(Mi[v], Mo[v])$.

[1] ex573.411.14

$$Mo[u] = \min_{v \in children(u)} Mi[v] + \sum_{w \in children(u), w \neq v} \min(Mo[v], Mi[v])$$

```
    Endif
Endfor
Return max(Mo[r], Mi[r]).
```

The algorithms clearly runs in polynomial time, as there are $3n$ subproblems for an n node tree, and each value associated with a subproblem of u of degree n_u can be determined in $O(n_u)$ time. So the total time is $O(\sum_u n_u) = O(n)$.

(b) We extend the algorithm for the case of bounded tree-width by having subproblems associated with the nodes of the tree-decomposition. For each node t of the tree-decomposition, let V_t be the subset of nodes corresponding to tree-node t, T_t the subtree rooted at t, and G_t the corresponding subgraph. Now for each disjoint sets $U, W \subset V_t$ we will define a subproblem and have $OPT(t, U, W)$ the minimum weight of a set in G_t that contains exactly the nodes U in V_t and covers all nodes of G_t except possibly not dominating the a subset of W. Recall that in the case of a tree, the recurrence for $OPT_{out}(u)$ was a little awkward, as we needed to select a child of u that is covering u. Here we would need to do this for each node in $V_t - (U \cup W)$. To make this simpler to write, we will define more subproblems. Let t be a node of the tree decomposition, and let $t_1, \ldots, t_d$ be its children, then we define subproblems $OPT(t, i, U, W)$ for each $0 \leq i \leq d$ to be the minimum weight of a set in the graph corresponding to the union of subtrees $T_{t_1}, \ldots, T_{t_i}$ and the set V_t that contains exactly the nodes U in V_t and dominates all nodes of this subgraph except possibly not dominating the a subset of W.

So if d_r is the degree of the root, then the optimum value we are looking for is $\min_{U \subset V_r} OPT(r, d_r, U, \emptyset)$ For any node t we have $OPT(t, 0, U, W) = \sum_{u \in U} c(u)$ if U dominates the nodes $V_t - W$, and ∞ otherwise. This defines the values at the leafs.

Given the subproblems, we will get the value at a node t using the values for smaller i, and the values at the children of t as follows. For a set U we will use the notation $c(U) = \sum_{u \in U} c(u)$, and $\delta(U)$ is the set dominated by U. Let t be a node of the tree decomposition and $t_1, \ldots, t_d$ its children, let n_i be the degree of child i.

(2) *The value of $OPT(t, i, U, W)$ for $i \geq 1$ is given by the following recurrence:*

$$\begin{aligned} OPT(t, i, U, W) &= c(U) + \min_{U_i \subseteq V_{t_i} : U_i \cap V_t = U \cap V_{t_i}} (OPT(t, i-1, U, W \cup \delta(U_i)) \\ &\quad + OPT(t_i, n_i, U_i, (\delta(U) \cup W) \cap V_{t_i}) - c(U \cap V_{t_i})) \end{aligned}$$

For a tree-decomposition of width k there are 3^{k+1} subproblems associated with a (t, i) pair, and there are n such pairs if the tree is of size n. Computing each value takes only $O(1)$ time, so the total time required is $O(3^k n)$.

Let $(T; \{V_t | t \in T\})$ be the given tree decomposition rooted at r. There are k $(s_i; t_i)$ terminal pairs. We focus on the tree-width 2 case. For convenience, we assume that there are no two pieces V_{t_1} and V_{t_2} where (t_1, t_2) is an edge and $V_{t_1} \subset V_{t_2}$. Consider the subgraph G_t. Note that there can be at most one i for which P_i both enters and leaves G_t, since any such path uses up at least 2 vertices of V_t. Note also that there can be at most 3 s_i-t_i terminal pairs that have one end in G_t and the other one outside (as the paths connecting such pairs must go through V_t.

- If there are 3 such pairs, that each node $v \in V_t$ must be connected via disjoint paths to one of them, and terminal pairs inside G_t must connect via paths inside G_t. There are $O(1)$ cases here to consider depending on which of the nodes in V_t is used to connect which of the 3 separated terminal pairs.

- If there are 2 such terminal pairs, than of the at most 3 nodes in V_t 2 must be connected via disjoint paths to one of them, the third is either not used in any of the paths or is used by path connecting two terminals s_i and t_i inside, or two terminals outside G_t. Now there are $O(k)$ cases to consider, depending on which terminal pair i is using the extra node.

- If there is only one such pair, than we can have one pair i with both terminals inside or outside of G_t, that uses 2 nodes in V_t, or the one path leaving G_t, can leave, come back and leave again, or one or two paths can use just one node in V_t while having both terminals inside or both outside of G_t. There are $O(k^2)$ cases to consider here.

- If there are no such pairs, than one path can use 2 or 3 nodes in V_t, or multiple paths can use one node each. Now there are $O(k^3)$ cases to consider.

We define multiple subproblems for each t according to the possibilities discussed above. For the at most 3 nodes in V_t there are at most $O(k^3)$ possible cases. This defines $Ok^3n)$ subproblems. The value of a subproblem is simply 0 or 1 (or true or false) depending whether or not there are disjoint paths in G_t that satisfy the state of the nodes in V_t corresponding to the subproblem, that is, connect each $v \in V_t$ to the terminal in question inside G_t (and possibly connect the two nodes in V_t to each other, if needed), via disjoint paths inside G_t. The desired disjoint paths exists if and only if the value of one of the subproblems that connects all terminal pairs within the subgraph G_r (which is the whole graph).

Given values for all the subproblems associated with the children $t_1, \ldots, t_d$ of a node t, we want to get the value of the given subproblem efficiently. To do this consider a node $v \in V_t$, the subproblem under question wants a particular paths P_i to go through this vertex in $O(d)$ time.

[1] ex209.650.476

Checking whether G is 1- or 2-colorable is easy. For $k = 3, 4, \ldots, w+1$, we test whether G is k-colorable by dynamic programming. We use notation similar to what we used in the Maximum-Weight Independent Set problem for graphs of bounded tree-width. Let $(T, \{V_t : t \in T\})$ be a tree decomposition of G. For the subtree rooted at t, and every coloring χ of V_t using the color set $\{1, 2, \ldots, k\}$, we have a predicate $q_t(\chi)$ that says whether there is a k-coloring of G_t that is equal to χ when restricted to V_t. This requires us to maintain $k^{(w+1)} \leq (w+1)^{(w+1)}$ values for each piece of the tree decomposition.

We compute the values $q_t(\chi)$ when t is a leaf by simply trying all possible colorings of G_t. In general, suppose t has children $t_1, \ldots, t_d$, and we know the values of $q_{t_i}(\chi)$ for each choice of t_i and χ. Then there is a coloring of G_t consistent with χ on V_t if and only if there are colorings of the subgraphs $G_{t_1}, \ldots, G_{t_d}$ that are consistent with χ on the parts of V_{t_i} that intersect with V_t. Thus we set $q_t(\chi)$ equal to *true* if and only if there are colorings χ_i of V_{t_i} such that $q_{t_i}(\chi_i) = true$ and χ_i is the same as χ when restricted to $V_t \cap V_{t_i}$.

[1] ex897.854.812

We build the following bipartite graph G. There is a node u_i for each variable x_i, a node v_j for each clause C_j, and an edge (u_i, v_j) whenever x_i appears in C_j.

We first note that since each variable appears three times, and each clause has length three, the number of nodes on the left side of G equals the number of nodes on the right side. More strongly, we claim that G in fact has a perfect matching.

This is a consequence of a more general claim: that if every node in a bipartite graph G has the same degree d, then G has a perfect matching. (Here $d = 3$.) Indeed, if G does not have a perfect matching, then by Hall's Theorem it has a set A on the left side for which $|A| < |\Gamma(A)|$, where $\Gamma(A)$ denotes the set of neighbors of any node in A. But A has $d|A|$ nodes coming out of it, and $\Gamma(A)$ can only absorb $d|\Gamma(A)|$ of them, so this is not possible.

Consider the perfect matching in the graph G we constructed from the *3-SAT* instance. For each variable, we set it in a way that satisfies the clause it is matched to. In this way, we satisfy the full collection of clauses.

[1] ex592.206.332

We root the tree at some node r and associate a subtree T_v with each node $v \in T$, and let n_v denote the number of nodes in the subtree T_v. A solutions for the graph partitioning problem may split the subtree T_v in any ratio, however, it is not hard to see that of (A, B) is the maximum weight cut splitting the tree T into two equal sides, and assume $v \in A$ and $|A \cap T_v| = k$, then the induced partition of T_v by having $A_v = A \cap T_v$, and $B_v = A \cap T_v$ is the maximum weight cut separating T_v into two sides, where the side containing v has size k. This is true as the only interaction of the partition of T_v with the remaining graph is through node v. More precisely, assume it is false, and consider the optimal such cut (A', B'). Replacing A_v by A' keeps all edges across the cut, expect replacing edges that cross the (A_v, B_v) cut with edges that cross the (A', B') cut, and hence it results in a cut of larger weight. This contradiction proves the claim.

Given the above observation, we define subproblems for each subtree T_v and each integer $k \leq n_v$, where $OPT(v, k)$ is the maximum cut of the subgraph T_v separating T_v into two sides where the side containing v has size k. The final answer is $OPT(r, n/2)$ if $n = n_r$ is the number of nodes in T.

We will use $c(u, v)$ as the cost or weight of the edge $e = (v, w)$ of the tree. For a leaf v we have $k = 1$ as the only possible value and $OPT(v, 1) = 0$. Now consider $OPT(v, k)$ for some non-leaf v. The side A containing v must contain $k - 1$ nodes in addition to node v. If v has only one child u then we need to consider two cases depending on whether or not the child u is on the same side of the cut as v, so we get that in this case

$$OPT(v, k) = \max(opt(u, k-1), c(v, u) + OPT(u, n_u - k + 1)).$$

If v has two children u and w, then we will consider all possible ways of dividing these $k - 1 = \ell_1 + \ell_2$ nodes between the two subtrees rooted at the two children of v. For each such division, there are further case depending whether or not the root of these subtrees is on the same side of the cut as v. We get the following recurrence in this case

$$\begin{aligned} OPT(v, k) &= \max_{0 \leq \ell_1 \leq k-1, \ell_2 = k-1-\ell_1} (\max(OPT(u, \ell_1), c(v, u) + OPT(u, n_u - \ell_1)) \\ &\quad + \max(OPT(w, \ell_2), c(v, w) + OPT(w, n_w - \ell_2)). \end{aligned}$$

There are $O(n^2)$ subproblems. We can compute the values of the subproblems starting at the leaves, and gradually considering bigger subtrees. This recurrence allows us to compute the value of a subproblem in $O(1)$ time given the values of the subproblems associated with smaller trees. So the total time required is $O(n^2)$.

[1] ex34.58.909

(a) Let $\{w_1, w_2, w_3\} = \{1, 2, 1\}$, and $K = 2$. Then the greedy algorithm here will use three trucks, whereas there is a way to use just two.

(b) Let $W = \sum_i w_i$. Note that in *any* solution, each truck holds at most K units of weight, so W/K is a lower bound on the number of trucks needed.

Suppose the number of trucks used by our greedy algorithm is an odd number $m = 2q+1$. (The case when m is even is essentially the same, but a little easier.) Divide the trucks used into consecutive groups of two, for a total of $q + 1$ groups. In each group but the last, the total weight of containers must be *strictly* greater than K (else, the second truck in the group would not have been started then) — thus, $W > qK$, and so $W/K > q$. It follows by our argument above that the optimum solution uses at least $q + 1$ trucks, which is within a factor of 2 of $m = 2q + 1$.

[1] ex667.592.236

(a) For each protein p, we define a set S_p consisting of all proteins similar to it; we do this by simply enumerating all proteins q for which $d(p,q) \leq \Delta$. With respect to these sets, a representative set $R \subseteq P$ is simply a set for which $\{S_p : p \in R\}$ if a set cover for P.

Thus, to approximate the size of the smallest representative set, we can use the approximation algorithm for Set Cover from this chapter, obtaining an approximation guarantee of $O(\log n)$.

(b) The problem with using the approximation algorithm for Center Selection is that we'd obtain a set R of proteins for which every protein is within distance 2Δ of some element of R. But this doesn't satisfy the requirements for a representative, which stipulated that every protein had to be within distance Δ of some element of R.

[1] ex815.903.104

(a) Let $a_1 = 1$ and $a_2 = 100$, and consider the bound $B = 100$. Only a_1 will be chosen, while an optimal solution would choose a_2.

(b) In fact, this can be done in $O(n)$ time. We first go through all the numbers a_i and delete any whose value exceeds B — such numbers cannot be used in any solution, including the optimal one, so we have not changed the value of the optimum by doing this.

We then go through the numbers $a_1, a_2, \ldots, a_n$ in order until the sum of numbers we've seen so far first exceeds B. Let a_j be the number on which this happens. Thus we have $\sum_{i=1}^{j} a_i \geq B$, but $\sum_{i=1}^{j-1} a_i \leq B$ and also $a_j \leq B$. Thus, one of the sets $\{a_1, a_2, \ldots, a_{j-1}\}$ or $\{a_j\}$ is at least $B/2$ and at most B; we select this set as our solution. Since the optimum has a sum of at most B, our solution is at least half the optimal value.

[1] ex650.691.264

Note that in case when all sets B_i have exactly 2 elements (i.e. $b = 2$), the Hitting Set problem is equivalent to the Vertex Cover problem (two-element sets B_i correspond to edges). In the chapter we saw two approximation algorithm for Vertex Cover; here we generalize the one based on linear programming to arbitrary b.

Consider the following problem for Linear Programming:

$$\begin{array}{lll} \text{Min} & \sum_{i=1}^{n} w_i x_i & \\ \text{s.t.} & 0 \le x_i \le 1 & \text{for all } i = 1, \dots, n \\ & \sum_{i:a_i \in B_j} x_i \ge 1 & \text{for all } j = 1, \dots, m \text{ (all sets are hit)} \end{array}$$

Let x be the solution of this problem, and w_{LP} is a value of this solution (i.e. $w_{LP} = \sum_{i=1}^{n} w_i x_i$).

Now define the set S to be all those elements where $x_i \ge 1/b$:

$$S = \{a_i \mid x_i \ge 1/b\}$$

(1) *S is a hitting set.*

Proof. We want to prove that any set B_j intersects with S. We know that the sum of all x_i where $a_i \in B_j$ is at least 1. The set B_j contains at most b elements. Therefore some $x_i \ge 1/b$, for some $a_i \in B_j$. By definition of S, this element $a_i \in S$. So, B_j intersects with S by a_i. ■

(2 *The total weight of all elements in S is at most $b \cdot w_{LP}$.*

Proof. For each $a_i \in S$ we know that $x_i \ge 1/b$, i.e., $1 \le bx_i$. Therefore

$$w(S) = \sum_{a_i \in S} w_i \le \sum_{a_i \in S} w_i \cdot bx_i \le b \sum_{i=1}^{n} w_i x_i = bw_{LP}$$

■

(3 *Let S^* be the optimal hitting set. Then $w_{LP} \le w(S^*)$.*

Proof. Set $x_i = 1$ if a_i is in S^*, and $x_i = 0$ otherwise. Then the vector x satisfy constrains of our problem for Linear Programming:

$$\begin{array}{ll} 0 \le x_i \le 1 & \text{for all } i = 1, \dots, n \\ \sum_{i:a_i \in B_j} x_i \ge 1 & \text{for all } j = 1, \dots, m \text{ (because all sets are hit)} \end{array}$$

Therefore the optimal solution is not worse that this particular one. That is,

$$w_{LP} \le \sum_{i=1}^{n} w_i x_i = \sum_{a_i \in S} w_i = w(S^*)$$

■

Therefore we have a hitting set S, such that $w(S) \le b \cdot w(S^*)$.

[1] ex53.496.888

In the textbook we prove that

$$T - t_j \leq T^*.$$

where T is our makespan, t_j is a size of a job and T^* is the optimal makespan. We also proved that the optimal makespan is at least the average load, which is at least 300 in our case:

$$T^* \geq \frac{1}{m}\sum_j t_j \geq \frac{1}{10}3000 = 300.$$

We also know that $t_j \leq 50$. Therefore the ratio of difference between our makespan and the optimal makespan to the optimal makespan is at most

$$\frac{T - T^*}{T^*} \leq \frac{t_j}{T^*} \leq \frac{50}{300} = \frac{1}{6} \leq 20\%$$

[1] ex995.878.454

One way to do this works as follows: When each job arrives, we put it on the machine that currently ends the soonest. (Note that this determination involves taking into account the speeds of the machines.)

To give a bound on this algorithm, we first give some lower bounds on the optimum makespan T^*. The total time of all jobs is $\sum_j t_j$. Let

$$t = \frac{\sum_j t_j}{m + 2k}.$$

If jobs could be assigned to machines so that each slow machine had a set of jobs summing to less than t, and each fast machine had a set of jobs summing to less than $2t$, then we would have

$$\sum_j t_j < mt + 2kt = \sum_j t_j,$$

a contradiction. Thus, some machine runs for at least t time units, and hence

$$T^* \geq \frac{\sum_j t_j}{m + 2k}.$$

Also, we have

$$T^* \geq \frac{1}{2} t_j,$$

for every job j, since at best it runs on one of the fast machines.

Let $M(r)$ denote the set of jobs assigned to machine r. Consider a machine i that achieves the makespan, and let j be the last job to go on it. Let x denote the time it uses for all jobs before j. (This means that $\sum_{j \in M(i)} t_j$ is equal to x if it's a slow machine, and it is equal to $2x$ if it's a fast machine.) Then at the moment j is added, every slow machine s has $\sum_{j \in M(s)} t_j \geq x$, and and every fast machine f has $\sum_{j \in M(f)} t_j \geq 2x$. Thus we have $\sum_j t_j \geq mx + 2kx$, and hence $T^* \geq x$. Also, $2T^* \geq t_j$.

Since the makespan is achieved by i, it is at most $x + t_j \leq T^* + 2T^* = 3T^*$.

An alternate solution is to simply sort the jobs in decreasing order of size, and then run the Greedy-Balance algorithm as though all machines were slow. We know from the chapter that this would give a $\frac{3}{2}$-approximation if all machines really were slow. However, we are comparing to the optimum as though all its machines are slow; in reality, the optimum's makespan might be half as large as we think, since some of its machines are fast. Thus, this gives a 3-approximation.

[1] ex829.220.704

(a) We process the customers in an arbitrary order. At any given point in time, let V_j denote the total value of all customers who have been shown ad j. As we see each new customer, we show him or her the ad for which V_j is as small as possible.

Let s' denote the spread of this algorithm. We first claim that $s' \geq \overline{v}/2m$. To prove this, suppose that ad j is the one achieving the spread (i.e., $V_j = s'$), and let i be any other ad. Let c be the last customer to be shown ad i. Before c was shown ad i, the value of V_i was at most V_j (by the definition of our greedy algorithm), and so $V_i \leq V_j + v_c \leq V_j + (\overline{v}/2m)$ by our assumption about the maximum customer value. Thus, if $s' = V_j < \overline{v}/2m$, then

$$\overline{v} = \sum_j V_j < V_j + (m-1)(V_j + (\overline{v}/2m)) < mV_j + (\overline{v}/2m) < (\overline{v}/2m) + (\overline{v}/2m) = \overline{v},$$

a contradiction.

Next we claim that the optimum spread s satisfies $s \leq \overline{v}/m$. Indeed, the total customer value is $\overline{v}$, and there are m ads, so one must be allocated at most a customer value of $\overline{v}/m$.

Combining these two claims, we get $s \leq \overline{v}/m \leq 2s'$.

(b) Suppose the input begins with $N + m$ customers of value 1, for some very large N, and then $m/2$ customers of value 2. (Suppose m is even and N is divisible by m.) Then our greedy algorithm will produce a spread of $1 + N/m$, while the optimal spread is $2 + N/m$, obtained by grouping the final m customers of value 1 onto $m/2$ ads, and showing the remaining $m/2$ ads to the customers of value 2.

[1] ex43.640.595

The conjecture is true. Consider the assignment of jobs to machines in an arbitrary optimal solution, and order the jobs arbitrarily on each machine. We say that the *base height* of a job j is the total time requirements of all jobs that precede it on its assigned machine.

We order all jobs by their base heights (breaking ties arbitrarily), and we feed them to the Greedy-Balance algorithm in this order. (We will label the jobs $1, 2, \ldots, n$ according to this order.)

We claim the following by induction on r: after the first r jobs have been processed by Greedy-Balance, the set of machine loads is the same as the set of machine loads if we consider the assignment of these r jobs made by the optimal solution.

This is clearly true for $r = 1$, since one machine will have load t_1, and all others will have load 0. Now suppose it is true up to some r, with loads $T_1, \ldots, T_m$, and consider job $r+1$. Because we have sorted jobs by base height, job $r+1$ comes from the machine that, in the optimal solution, has load $\min_i T_i$. By the definition of Greedy-Balance, this is the machine on which job $r+1$ will be placed, giving it a load of $t_{r+1} + \min_i T_i$. This completes the induction step.

[1] ex286.347.713

We will use the following simple algorithm. Consider triples of T in any order, and add them if they do not conflict with previously added triples. Let M denote the set returning by this algorithm and M^* be the optimal three-dimensional matching.

(1) *The size of M is at least $1/3$ of the size of M^*.*

Proof. Each triple (a, b, c) in M^* must intersect at least one triple in our matching M (or else we could extend M greedily with (a, b, c)). One triple in M can only be in conflict with at most 3 triples in M^* as edges in M^* are disjoint. So M^* can have at most 3 times as many edges as M has. ■

[1] ex271.721.76

(a) If $v \notin S$, it must have never been chosen by the greedy algorithm. This means that it was deleted in some iteration by the selection of a node v': by the definition of the selection rule, this node v' must both be a neighbor of v, and have at least as much weight as v.

(b) Consider any other independent set T. For each node $v \in T$, we *charge* it to a node in S as follows. If $v \in S$, then we charge v to itself. Otherwise, by (a), v is a neighbor of some node $v' \in S$ whose weight is at least as large. We charge v to v'.

Now, if v is charged to itself, then no other node is charged to v, since S and T are independent sets. Otherwise, at most four neighboring nodes of no greater weight are charged to v. Either way, the total weight of all nodes charged to v is at most $4w(v)$. Since these charges account for the total weight of T, it follows that the total weight of nodes in T is at most four times the total weight of nodes in S.

[1] ex727.874.96

This means that our knapsack has capacity $(1+2\epsilon)W$. We throw out all items of weight exceeding W, since these can't be used in the solution we're comparing against.

We now round all remaining weights down to the nearest multiple of $\epsilon W/n$, and then multiply them all by $n/(\epsilon W)$. This means that all weights are now integers between 0 and n/ϵ. (Note that the items of weight less than $\epsilon W/n$ do get rounded down to 0, and yes, this means we will probably take them all, but as we'll see this is not a problem.)

So in time polynomial in n and $1/\epsilon$, by the dynamic programming algorithm for the knapsack problem with small weights, we can find the subset of (rounded) weight at most W which achieves the greatest value. The solution of (true) weight W and value V that was promised to exist must be available as an option, since its weight only went down, and since we find the best subset, we find one of value at least V. When we put all these items in our knapsack, each has a weight that may be up to $\epsilon W/n$ more than we thought (due to rounding down), so we use a weight of at most $W + n(\epsilon W/n) = W(1+\epsilon)$.

[1] ex662.412.328

We'll select the sites and the users they cover using the idea of the Set-Cover greedy algorithm. If a side s is used to cover the a subset U_s of users, then the average user cost is $(f_s + \sum_{u \in U_s} d_{us})/|U_s|$. The idea behind the greedy algorithm is to select the site s with a subset U_s that minimizes this quantity. First we need to argue that this minimum can be found.

(1) *Given a set R of uncovered users, and a site possible s, one can find the subset $U_s \subset U$ that minimizes the average cost $(f_s + \sum_{u \in U_s} d_{us})/|U_s|$ in polynomial time.*

Proof. Sort the users by increasing distance d_{us} from site s. The set U_s will be an initial set of this sorted sequence: $U_s = \{u \in R : d_{su} \leq \alpha\}$ for some value α. ∎

Now the algorithm will be analogous to the Set Cover greedy algorithm. We select sites s with subsets U_s by the above greedy rule: selecting the site and the set that minimizes the average cost of covering a new user. There is one more option to consider. Suppose T is the subset of sites already selected. For a site $s \in T$ we can add a new node $u \in R$ to U_s, covering the new user u at the cost of d_{us}. In the algorithm given below, we will also save the cost c_u at which user u got covered by the algorithm. These values will be used by the analysis.

```
Start with R = U and T = ∅.
While R is not empty
   Let c = min_{u∈R,s∈T} d_us
   Select s ∈ S−T, and set U_s ⊂ R that minimizes
          c' = (f_s + Σ_{u∈U_s} d_us)/|U_s|.
   If c' ≤ c then
      Select the site s and set U_s used to obtain c' above.
      Add s to T, and delete U_s from R.
      Set c_u = c' for all u ∈ U_s.
   Else
      Select s and u obtaining the first minimum.
      Add u to U_s,
      Set c_u = c.
Endwhile
```

First, note that if we select the set of sites T, and have each site $s \in T$ cover the users in U_s then we get a solution to the problem with total cost $\sum_{u \in U} c_u$. Also, the algorithm runs in polynomial time. It remains to show that this is an $H(n)$ approximation algorithm.

The proof of the approximation ratio follows very closely the proof for the set cover algorithm. Consider an optimum solution. Assume it contains a subset T^* of sites, and $s \in T^*$ is used to cover a set U_s^* of users. The cost of using s to cover U_s^* is $f_s + \sum_{u \in U_s^*} d_{us}$. We will want to compare the optimum's cost, and $\sum_{u \in U_s^*} c_u$, which is the cost our greedy algorithm paid for the users in U_s^*.

[1] ex37.588.671

(2) *Using the notation introduced above, and the costs defined by the algorithm, we have that* $\sum_{u \in U_s^*} c_u \leq H(d)(f_s + \sum_{u \in U_s^*} d_{us})$, *where* $d = |U_s^*|$.

Proof. For notational simplicity, let $C = f_s + \sum_{u \in U_s^*} d_{us}$. Consider the elements in U_s^* in the order the algorithm covered them. Assume they are $u_1, u_2, \ldots, u_d$. Consider the moment the algorithm covers the ith node u_i from U_s^*. There are two cases to consider.

Case 1 At this point of the algorithm $s \notin T$.

Case 2 At this point of the algorithm $s \in T$.

When the algorithm covered u_i it selected the smallest average cost. In Case 1 this implies that the cost c_{u_i} is at most the cost of selecting cite s with the set $U_s^* \cap R$, which is at most $c_{u_i} \leq C/(d - i + 1)$ (as $i - 1$ previously covered nodes are no longer in the set). In Case 2, this implies that $c_{u_i} \leq d_{us}$. Assume that Case 1 applies when the first k nodes are covered, and after that Case 2 applies (k may be equal to d). Now summing all costs in U_s^* we get that

$$\sum_{u \in U_s^*} c_u \leq C/d + C/(d-1) + \ldots C/(d-k+1) + \sum_{i > d} d_{u_i,s}.$$

Now if $d = k$ then the upper bound on the cost is $H(d)C$ as claimed. If $k < d$ then note that the costs $\sum_{i>d} d_{u_i,s}$ is bounded by C, and so we also can bound the total cost by $H(d)C$. ∎

Now we are ready to prove that the algorithm is an $H(n)$ approximation algorithm. Let T^* and U_s^* be the optimal solution. The total cost of the solution is $\sum_{s \in T^*}(f_s + \sum_{u \in U_s^*} d_{us})$. We use the above Lemma to bound each term of the cost, and upper bound $H(d)$ by $H(n)$ for each set U_s^* in the optimum, to get the following.

$$OPT = \sum_{s \in T^*}(f_s + \sum_{u \in U_s^*} d_{us}) \leq \sum_{s \in T^*} H(n) \sum_{u \in U_s^*} c_u = H(n) \sum_{u \in U} c_u,$$

where the last sum is the algorithm's cost as claimed by the first Lemma.