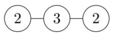
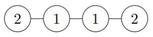
1.
   - 
   - 
   - A maximum weight independent set in a path of n vertices either contains the last vertex v_n or not. If it contains v_n then it also contains a maximum weight independent set from the vertices $v_1, v_2, \ldots, v_{n-2}$ (it cannot contain v_{n-1}). If it doesn't contain v_n, then it is a maximum weight independent set from the vertices $v_1, v_2, \ldots, v_{n-1}$. Let $A[i]$ be the weight of a maximum weight independent set from vertices $v_1, v_2, \ldots, v_i$. $A[0] = 0$, $A[1] = w(v_1)$, and $A[n]$ is the weight of the independent set we seek. For $i \geq 2$,

$$A[i] = \max\{A[i-1], A[i-2] + w(v_i)\}.$$

Calculating each entry in $A[]$ takes constant time, given that we calculate $A[i]$ in order of increasing i, so the running time is $O(n)$. To produce the independent set, we record the choice (say $i-1$ or $i-2$) we make to maximize the value of $A[i]$ in an auxiliary array $B[i]$ (during the calculation of $A[i]$) and then, after we calculate $A[n]$, we produce the independent set recursively as:

$$S(n) = \begin{cases} S(n-1) & \text{if } B[n] = n-1 \\ S(n-2) \cup v_n & \text{if } B[n] = n-2 \end{cases}$$

where $S(0) = \emptyset$ and $S(1) = \{v_1\}$.

2. Let $F[i,j]$ (for $i \leq j$) be the set of characters that can result from parenthesizing $x_i x_{i+1} \ldots x_j$. $F[i,i] = \{x_i\}$ for $i = 1 \ldots n$. $F[i,j] = \bigcup_{k=i}^{j-1} F[i,k] \cdot F[k+1,j]$ where $X \cdot Y = \{x \cdot y | x \in X, y \in Y\}$ and $\cdot$ is multiplication from the table. The answer is determined by whether $a \in F[1,n]$.

   $F[1,n]$ can be be calculated in $O(n^3)$ time. Start with the diagonal entries. Calculate $F[i,j]$ in $O(d)$ time where $j - i = d > 0$ only after all entries $F[i,j]$ with $0 \leq j - i < d$ have been calculated. Since d is at most $n-1$, all $O(n^2)$ entries $F[i,j]$ with $j \geq i$ can be calculated in time $O(n^3)$.

3. Let $F[i,b]$ be the value of the maximum value subset from items $1, 2, \ldots, i$ that has weight at most b. We are interested in calculating $F[n,W]$. To obtain a maximum value subset from items $1, 2, \ldots, i$ that has weight at most b, we can either take item i into our knapsack, provided $w_i \leq b$, and then find a maximum value subset from items $1, 2, \ldots, i-1$ to pack in the space that remains in our knapsack (getting value $F[i-1, b-w_i] + v_i$); or we can leave item i, and find a maximum value subset from items $1, 2, \ldots, i-1$ of weight at most b (getting value $F[i-1,b]$). So,

$$F[i,b] = \begin{cases} \max\{F[i-1, b-w_i] + v_i, F[i-1,b]\} & \text{if } w_i \leq b \\ F[i-1,b] & \text{otherwise.} \end{cases}$$

The base cases are when we have run out of items: $F[0,b] = 0$ for all $0 \leq b \leq W$; or when we have run out of capacity: $F[i,0] = 0$ for all $0 \leq i \leq n$. The table $F[,]$ is of size $O(nW)$ and it takes constant time to fill in entry $F[i,b]$ as long as we fill in the table in increasing order of i. To obtain the actual set of items to take, we can record the choice (take or leave) made to maximize $F[i,b]$ (in a table $B[i,b]$) and reconstruct the solution using recursion:

$$T(n,W) = \begin{cases} T(n-1, W-w_n) + v_n & \text{if } B[n,W] = \text{take} \\ T(n-1,W) & \text{if } B[n,W] = \text{leave} \end{cases}$$

4. Let $b_{ij} = M - j + i - \sum_{k=i}^{j} \ell_k$ be the number of blanks at the end of a line that contains words i through j. Define the cost of such a line to be

$$c_{ij} = \begin{cases} \infty & \text{if } b_{ij} < 0 \\ 0 & \text{if } b_{ij} \geq 0 \text{ and } j = n \\ b_{ij}^3 & \text{if } b_{ij} \geq 0 \text{ and } j < n. \end{cases}$$

Calculating b_{ij} and c_{ij} for all $1 \leq i < j \leq n$ can be done in $O(n^2)$ time, since we can calculate the partial sums $\sum_{k=i}^{j} \ell_k$ for all $i < j$ in $O(n^2)$ time. (How?)

To print the paragraph neatly, we rely on dynamic programming. The best way to print words 1 to j is to print words 1 to $i-1$ optimally and then to print words i through j on a new last line, for some value of $i \leq j$. Let $C[j]$ be the minimum cost of printing words 1 to j.

$$C[j] = \min \{C[i-1] + c_{ij}\}$$

where $C[0] = 0$. The computation of $C[j]$ takes $O(j)$ time (assuming $C[i-1]$ and c_{ij} have already been calculated for $i \leq j$). So to calculate $C[n]$, which requires calculating $C[j]$ for $j \leq n$, takes $O(n^2)$ time. By recording the choice of i used to calculate $C[j]$, we can produce the actual paragraph recursively. (How?)

The space required by this algorithm is $O(n^2)$ (to store c_{ij}). We could reduce the space to $O(n)$ without affecting the running time by keeping c_{ij} for one value of j and all $i \leq j$ until we finish calculating $C[j]$ and then reusing the space to calculate (in time $O(j)$) $c_{ij'}$ for $j' = j+1$ and all $i \leq j+1$ to be ready for calculating $C[j+1]$.

Note that if $M < n$ we can get a running time of $O(nM)$ since we know $c_{ij} = \infty$ for $j - i > M$.

1. (a) Let f be a flow with size$(f) = 1$. The flow f is composed of a number of simple paths $P_1, P_2, \ldots, P_k$ ($k \geq 1$) from s to t carrying flows of size $a_1, a_2, \ldots, a_k$ respectively, where $\sum_{i=1}^{k} a_i = 1$. (These paths are not necessarily edge or vertex disjoint.) For each edge e, $\sum_{P_i \ni e} a_i = f_e$ (the notation $P_i \ni e$ means paths P_i that contain the edge e) so

$$\sum_e \ell_e f_e = \sum_e \ell_e \left(\sum_{P_i \ni e} a_i \right) = \sum_{i=1}^{k} a_i \left(\sum_{e \in P_i} \ell_e \right) = \sum_{i=1}^{k} a_i \ell(P_i)$$

where $\ell(P_i)$ is the length of path P_i. Since $\sum_{i=1}^{k} a_i \ell(P_i)$ is an average of st-path lengths, the average is at least the minimum st-path length. Thus, minimizing $\sum_e \ell_e f_e$ is equivalent to finding the shortest path P_i from s to t and assigning $a_i = 1$ (or equivalently $f_e = 1$ for all $e \in P_i$).

(b) The variables are f_e.

$$\begin{aligned} \min \sum_e \ell_e f_e & \\ \sum_w f_{vw} - \sum_u f_{uv} = 0 & \quad \text{for each vertex } v \neq s,t \\ \sum_w f_{sw} - \sum_u f_{us} = +1 & \\ \sum_w f_{tw} - \sum_u f_{ut} = -1 & \\ f_e \geq 0 & \quad \text{for each edge } e \end{aligned}$$

(c) The dual has one variable for every constraint. Since each constraint corresponds to a vertex v, call these variables x_v for v a vertex in the graph G. After multiplying each constraint by its corresponding variable and summing up the constraints, we have for every edge (u,v) in G, the coefficient of f_{uv} is $x_u - x_v$, since x_u multiplies all outgoing edges of u and $-x_v$ multiplies all incoming edges to v. To be a lower bound on $\sum_e \ell_e f_e$, we want each such coefficient to be at most ℓ_{uv}. To be the best lower bound on $\sum_e \ell_e f_e$, we want to maximize the right-hand side of the summed constraints, i.e., $x_s - x_t$.

2. (a) $\max \sum_{uv \in E} x_{uv}$
   $\sum_{v \in V} x_{uv} \leq 1$ for all $u \in U$
   $\sum_{u \in U} x_{uv} \leq 1$ for all $v \in V$
   $x_{uv} \geq 0$ for all $uv \in E$

   (b) $\min \sum_{u \in U} y_u + \sum_{v \in V} y_v$
   $y_u + y_v \geq 1$ for all $uv \in E$
   $u_{..} > 0$ for all $u \in U$
   every edge in G has an endpoint in the set, and takes value 0 otherwise. This problem is called (MINIMUM) VERTEX COVER and is NP-hard for general graphs, but solvable in polynomial time using network flow for bipartite graphs.
   Can you see how the minimum capacity cut can be used to find the VERTEX COVER?

3. This shows how much *Row* (the row player) wins.

	5	10
5	+5	-5
10	-10	+10

$$\begin{aligned} \max z & \\ z \leq 5x_1 - 10x_2 & \\ z \leq -5x_1 + 10x_2 & \\ x_1 + x_2 = 1 & \\ x_1, x_2 \geq 0 & \end{aligned}$$

ability $b/(a+b)$
oin to hide with

4. The payoff matrix for this game is:

$$\begin{array}{c|ccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & \ldots \\ \hline 1 & 0 & -1 & +1 & +1 & +1 & +1 & \ldots \\ 2 & +1 & 0 & -1 & +1 & +1 & +1 & \ldots \\ 3 & -1 & +1 & 0 & -1 & +1 & +1 & \ldots \\ 4 & -1 & -1 & +1 & 0 & -1 & +1 & \ldots \\ 5 & -1 & -1 & -1 & +1 & 0 & -1 & \ldots \\ 6 & -1 & -1 & -1 & -1 & +1 & 0 & \ldots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots \end{array}$$

For $n = 2$, player *Row* should choose 2. The best response for *Col* is to choose 2 as well, for a game value of 0. Why is this optimal? *Row* cannot hope for a better (larger) value than 0 since *Col* has a strategy (choose 2) for which *Row* can achieve at most value 0. This is trivially a smallest optimal strategy.

For $n = 3$, player *Row* should choose 1, 2, or 3 with equal probability, for a game value of 0. *Row* cannot hope for a better value since *Col* has a strategy, also $(1/3, 1/3, 1/3)$, for which *Row* can achieve at most value 0. This is a smallest optimal strategy since if *Row* plays on only a subset of $\{1,2,3\}$, *Col* has a pure strategy that insures a negative game value. For example, if *Row* plays a mixed strategy on rows $\{1,3\}$, *Col* plays 1 and insures a negative value.

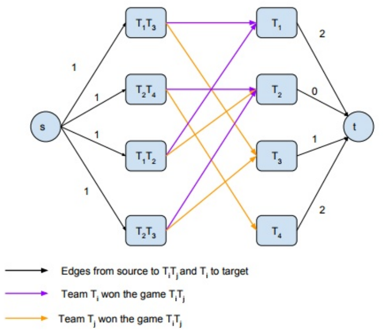
For $n > 3$, player *Row* should choose 1, 2, or 3 with equal probability, for a game value of 0. If *Row* plays a strategy involving more integers, *Col* will play 1, 2, or 3 with equal probability and insure a negative value. As described for the case $n = 3$, this is a smallest optimal strategy for *Row*.

$L = (A, T_1), (A, T_3), (A, T_4), (T_1, T_3), (T_2, T_3), (T_2, T_4), (T_1, T_2)$

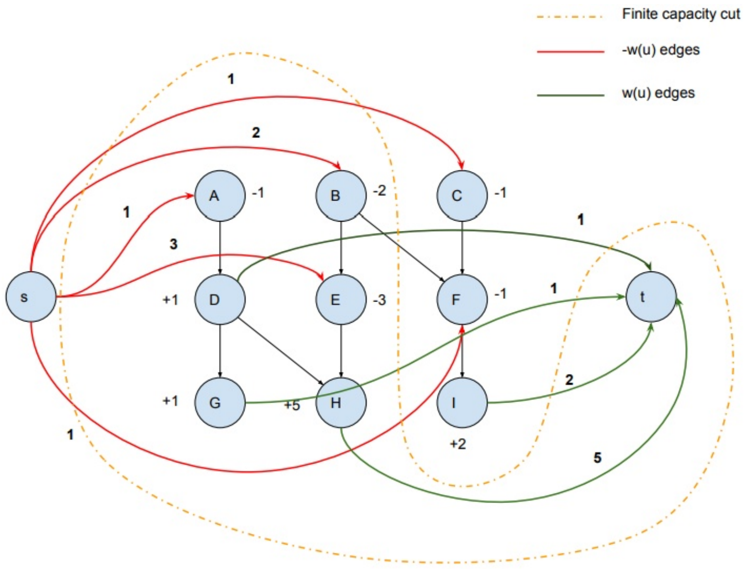
Then, we compute w and w_i

team	#wins	$w - w_i$
A	3	w = 6, *i.e.* 3 + winning $(A, T_1), (A, T_3), (A, T_4)$
T_1	4	2
T_2	6	0
T_3	5	1
T_4	4	2

As all w_i satisfy step **4** of the algorithm, we need to create the network flow and compute max-flow f.



s
t
T_1T_3
T_2T_4
T_1T_2
T_2T_3
T_1
T_2
T_3
T_4
1
1
1
1
2
0
1
2
Edges from source to T_iT_j and T_i to target
Team T_i won the game T_iT_j
Team T_j won the game T_iT_j



$profit = 1$

$\sum_{u \in V | w(u) > 0} w(u) = 9$

$\sum_{u \notin U | w(u) > 0} w(u) = 2$

$\sum_{v \in U | w(v) < 0} w(v) = -6$

Therefore,

$$9 - 1 = 2 - (-6) \rightarrow 8 = 8$$

This example serves to express that maximizing the profit is the same as minimizing the loss and thus, there is a dual way to express the open-pit mining problem.