CSE202 Homework 3

Erik Buchanan, Matt Jacobsen, Noah Wardrip

June 5, 2008

Q1 - Grade maximization

We present a polynomial time dynamic programming solution to the grade maximization problem described below.

Formal specification of problem

Instance:

Given: $H \in Z^+$ hours, $n \in Z^+$ class projects, and a set of functions $\{f_i : i = 1, 2, \ldots n\}$, where $s = f_i(h)$ means that if h hours are spent on class project i, with $h \leq H$ and $h \in N$, a grade of $s \leq g$ will be received for class i, bounded by some maximum grade g.

Solution Space:

An array S that specifies how many (integer) hours to spend on each class's project, such that $\forall i \in \{1, 2, \ldots n\}, S[i] = h$ means spending h hours on the project for class i.

Constraints:

There are H total hours to allocate, so $\forall i \in \{1, 2, \ldots n\}, 0 \leq S[i] \leq H, S[i] \in N$ and $\sum_{i=1}^{n} S[i] \leq H$. The algorithm must complete in polynomial time with respect to n, g, and H.

Objective:

Maximize the average project grade. Equivalently, maximize the total grade across all classes, $\sum_{i=1}^{n} f_i(S[i])$, since the sum is simply n times the average.

Definitions:

In addition to the variables defined in the problem specification, we define the following variables used in the algorithm:

- $M[][]$ is the "memory" for the dynamic programing algorithm. It is a 2-dimensional array: each element $M[h]$ is the optimal solution to the grade maximization problem for h hours to allocate. Thus, $M[H] = S$ is the algorithm's solution for H hours.
- $TotalGrade[]$ is an array which stores the sum of the grades over all classes for a particular solution, which we are trying to maximize. Thus, $TotalGrade[h] = \sum_{c=1}^{n} f_c(M[h][c])$.
- $BestGrade$ is a variable that stores the best total grade found so far for a given value of h.
- $Grade$ is a temporary variable which stores the total value of a given allocation of hours to class projects.
- $BestK$ is the best previous solution found so far to which to allocate additional hours.
- $BestC$ is the best class project found so far to which to allocate additional hours.

Algorithm and Analysis:

Algorithm:

```
1  for c = 1...n do
2      M[0][c] ← 0;
3  end
4  TotalGrade[0] ← 0;
5  for h = 1...H do
6      BestGrade ← 0;
7      for k = 0...h − 1 do
8          for c = 1...n do
9              Grade = TotalGrade[k] − f_c(M[k][c]) + f_c(M[k][c] + h − k);
10             if Grade > BestGrade then
11                 BestGrade ← Grade;
12                 BestK ← k;
13                 BestC ← c;
14             end
15         end
16     end
17     M[h] ← M[BestK];
18     M[h][BestC] ← M[h][BestC] + h − BestK;
19     TotalGrade[h] ← BestGrade;
20 end
21 return M[H];
```

The algorithm recursively considers subproblems of the proposed problem in the fashion of dynamic programming. First, find the ideal allocation of the one hour among the n class projects. Then, to find the ideal allocation of two hours, there are two options: find the ideal class project to which to allocate an hour, in addition to the solution from the previous step, or allocate the two hours as a unit to a single class. These two differ because perhaps the first hour was most effective in class i, but one hour spent on the project for class j would yield very little benefit. The second hour might not be allocated to class j either, but perhaps two hours spent on class project j would achieve a grade much higher than any other option. Thus, hours allocated in varying sized groups must be considered.

The algorithm first initializes $M[0]$ to be accurate for no time spent on any class project, so that each subsequent solution may build on this case. Then, for an increasing number of hours, each integer value is considered. For each of the previously computed ideal allocations ($M[k]$), it considers each of the n class projects to which to allocate $h-k$ hours. The total grade for each allocation is computed from the previous total grade and the change in the grade for class project c, using $f_c()$. If this grade $Grade$ is better than the current best grade $BestGrade$, then this choice is recorded via $BestGrade$, $BestK$, and $BestC$. Then the next allocation option is considered.

Once all possible allocation options for h hours have been considered, the best is recorded in $M[h]$ and $TotalGrade[h]$. For each, we consider allocating $1, 2, \ldots h$ hours at a time to each class project, adding on to the ideal allocation for the other k hours. Finally, when we have computed the ideal allocation for all H hours, we return that allocation $M[H]$.

Correctness:

Proof. We prove that the algorithm above correctly solves the problem and that its solution is optimal. The proof that our algorithm terminates is given in the *Time Analysis* section below. We prove correctness by induction on each iteration of the outer loop where h goes from 1 to H.

`Base case:` The only valid solution which satisfies the constraints with $H = 0$ is the array S such that $\forall i \in \{1, 2, \ldots n\} S[i] = 0$. The algorithm initializes $M[0]$ to such an array, and if $H = 0$ the loop will be skipped, and $M[0]$ returned.

`Inductive step:` Assume $M[i]$ is a valid, optimal solution to the prolem where $H = i$ for all $0 \leq i < h$. We prove that $M[h]$ is an optimal solution where $H = h$, for any $h \geq 1$.

Lemma 1. *Every optimal solution for $H = h$ hours is an improvement on an optimal solution for $H = i$ hours, where $0 \leq i < h$. Improvement means it allocates at least as many hours to each class project as the previous solution did, and possibly more.*

Proof of Lemma 1. The statement is trivially true, because $M[0]$ is an optimal solution for $H = 0$, and every valid solution must assign at least 0 hours to each class project. □

Lemma 2. *For any optimal solution S for $H = h$ hours, there exists an optimal solution for $H = i$ hours, $M[i]$, for some $0 \leq i < h$, where $\forall j, 1 \leq j \leq n, j \neq k, M[i][j] = S[j]$ for some k. That is, there exists an optimal solution for i hours that allocates the same number of hours to each class project, except for one class project, to which it allocates fewer hours.*

Proof of Lemma 2. Proof by contradiction: Suppose that S is an optimal solution for $H = h$ hours, and that there does not exist an optimal solution for $0 \leq i < h$ hours that allocates the same number of hours to each class except for one. By Lemma 1, S must be an improvement on some optimal solution for fewer hours.

Let $M[j]$ be the solution which is optimal for the largest number of hours $j < h$, on which S is an improvement. Then S must have increased the allocation of hours to at least two classes, compared to $M[j]$. If we construct S', which improves upon $M[j]$ by selecting the most beneficial k allocation increases that S made over $M[j]$, then S' must also be an ideal allocation of k hours, with $j < j + k < h$. Because all allocation increases were selected from S, S must be an improvement on S'. This contradicts the assumption that $M[j]$ was the optimal solution for the largest number of hours less than h. □

Suppose that S is an optimal solution for $H = h$. By Lemma 1, S must be an improvement on $M[i]$ for some $0 \leq i < h$. By Lemma 2, S must be an improvement on some $M[i]$ where the allocation of hours only increased for one class project. The algorithm compares all ideal solutions for fewer than h hours, adding in the remaining unallocated hours to each class, and chooses the best such allocation over all previous solutions and allocating the remaining hours to all possible classes. Thus, we must consider S when choosing $M[h]$. Because we choose the allocation with the largest total grade, the total grade earned by $M[h]$ must be at least the grade earned by S. This completes the inductive step proof.

Therefore, at the end of the algorithm, $M[H]$ is an optimal solution for H hours, which maximizes the total (and therefore average) grade across all class projects, according to the set of functions f_i. □

Time Analysis:

The algorithm first initializes $M[0]$ and $TotalGrade[0]$, which assigns $n + 1$ values in $O(n)$ time. The body of the algorithm contains three nested loops. The first loop iterates over M, which is an array of length H, so it iterates exactly H times. The next loop iterates over each of the elements of M that have already been filled in, which iterates $O(H)$ times. The inner loop iterates over each of the n class projects, and performs a constant amount of work (assuming the function f_c completes in constant time). Thus, this loop iterates exactly n times per execution, which is $O(n)$. The work done at the end of the outside loop is also performed in constant time, as is the return statement. Thus, the overall runtime of the algorithm is $O(n + H \times H \times n) = O(H^2 n)$.

As described above, each of the loops iterates a fixed number of times (where the second loop iterates h times, with $h \leq H$ in every iteration). Therefore, the algorithm must terminate. It completes in $O(H^2 n)$ time.

Q2 - Number of rectangular prisms

We give a dynamic programming polynomial time algorithm that given integers d and m returns the number of rectangular prisms with integer valued sides $s_1, ...s_d$ and diagonal D such that $D^2 = m$.

```
1  L is a m x d matrix, all values are initially 0;
2  SQ = ∅;
3  for r = 1 to ⌊√m⌋ do
4      SQ = SQ + {r^2};
5  end
6  for each sq ∈ SQ do
7      L[sq, 1] = 1;
8  end
9  for c = 2 to d do
10     for each sq ∈ SQ do
11         for r = 1 to m − sq do
12             if L[r, c − 1] > 0 then
13                 L[r + sq, c] = L[r, c − 1] + 1;
14             end
15         end
16     end
17 end
18 return L[m, d];
```

The algorithm populates an initially zero valued array L with the number of different rectangular prisms that can be constructed for each entry. The first loop on line 3 calculates the set SQ which contains the square of each number from 1 to $\lfloor\sqrt{m}\rfloor$. SQ contains the square of every integer value that any side s_i could have. The next loop on line 6 initializes the first column of L to contain the values of each square in SQ. This corresponds to the rectangular prisms that could be constructed using 1 side where $D^2 \leq m$. The 3 nested loop structure starting on line 9 populates the rest of the array by column and then row. For each square $sq \in SQ$ the algorithm increases the prism count of any entry > 0 in the previous column, placing the result in the row corresponding to the sum of the squares from the previous column and sq. The inner loop on line 11 only iterates up to $m - sq$ because any value greater than $m - sq$ will exceed the matrix bounds when sq is added. Once the algorithm completes, the entry at row r and column c represents the number of prisms that can be constructed with c sides and diagonal $D^2 = r$.

We show that the algorithm terminates. The algorithm contains five distinct loops, but all iterate for a finite number of times. The loop on line 3 iterates for at most $\lfloor\sqrt{m}\rfloor$ times. The loop on line 6 iterates $|S| \leq \lfloor\sqrt{m}\rfloor$ times. The loop on line 9 iterates at most d times. The inner loop on line 10 iterates for $|S| \leq \lfloor\sqrt{m}\rfloor$ also and the inner loop on line 11 iterates for at most m times. Thus all loops in the algorithm iterate a finite number of times and the algorithm terminates.

We now show that the algorithm is correct. First we show that the entries the matrix L contains after the algorithm is run contains the number of prisms that can be constructed as desired. Initially L is zero filled. Consider the first loop on line 3. This loop calculates the set SQ which contains the square of each number from 1 to $\lfloor\sqrt{m}\rfloor$. This loop iterates only up to $\lfloor\sqrt{m}\rfloor$ because by the definition of the diagonal for a rectangular prism, any integer larger than $\lfloor\sqrt{m}\rfloor$ would have a squared diagonal exceeding m and thus could not be a value of any side s_i. Now, SQ contains the square of every integer value that any side s_i could take. The next loop on line 6 populates the first column with a 1 for each row corresponding to the value of the squares in SQ. Afterwards, the entries in L's first column contain the number of prisms that can be constructed using 1 side. We now show via induction on the number of columns in L that the algorithm

populates each column with entries containing the number of prisms that can be constructed using m and d.
Base case: for $c = 1$. We have just shown that each entry in the first column contains a 0 if no prism can be constructed or a 1 if a prism can be constructed using one of the values in SQ.
Inductive step: assume L contains the number of prisms that can be constructed for entries through m and $c-1$. Now when the algorithm populates column c, it iterates through each square sq in SQ. For each sq, the algorithm then checks every entry in the previous column $c-1$. If the entry has a value $v > 0$ at position r, the algorithm sets the value for the entry at position $r + sq$ in the current column c to $v + 1$. This is effectively taking the number of prism solutions, v with $c-1$ sides and a squared diagonal of r, then adding another side of length $\sqrt{sq}$ to make a new set of prisms with cardinality $v + 1$ that have c sides and a squared diagonal of $r + sq$. Because this is done for every value sq in SQ, every possible value for the additional side is considered and the c^{th} column contains entries with the number of prisms that can be constructed with c sides. By the inductive hypothesis, we see that the algorithm populates all columns of L with entries m, d with the number of rectangular prisms with integer valued sides $s_1, ...s_d$ and diagonal D such that $D^2 = m$.

Now we show that the number of number of rectangular prisms with integer valued sides $s_1, ...s_d$ and diagonal D such that $D^2 = m$ is correctly found by the algorithm and stored in $L[m, d]$. Any prism with d sides can be constructed with integer side values at most $\lfloor\sqrt{m}\rfloor$. If any side were greater than $\lfloor\sqrt{m}\rfloor$, it alone would exceed the squared diagonal m for the prism, by the definition of the diagonal of a rectangular prism. Therefore, the fact that the algorithm only considers integer values for prism sides who's square sq is $\leq m$ does not affect its ability to find all possible prisms. Now, a prism with one side can be constructed using just 1 integer and thus the squared diagonal is exactly the square of the side. This is represented exactly by the first column of L as the result of lines 6 - 8. Prisms with more sides can be constructed from prisms of one fewer side by taking the number of prisms v that can be made using $c-1$ sides with a squared diagonal value r, and adding a side with value $\sqrt{sq}$ to make $v+1$ prisms with c sides and a squared diagonal of $r+sq$. Indeed this is exactly what happens in lines 10 - 17. The prisms that can be constructed of length $c-1$ are searched. If a number of prisms $v > 0$ can be made with $c-1$ sides, having diagonal squared length r, the algorithm adds an additional side of length $\sqrt{sq}$ to create a new set of prisms with cardinality $v + 1$. The new prisms have diagonal length $r + sq$ because of the added side. As such, the new number of prisms $v + 1$ is placed in row $r + sq$ corresponding to the new diagonal length. The algorithm does this for every possible value of sq in SQ and for d prism sides. Thus at the end of the algorithm, the number in L at entry m, d corresponds to the number of possible prisms that can be constructed with d integer valued sides and a squared diagonal m.

We now show the running time of the algorithm is polynomial in the size of m and d. The initialization of L on line 1 can be done by setting each element in the matrix to 0. This takes $O(md)$ time. The loop on line 3 iterates at most $O(m^{0.5})$ times and performs a constant number of operations within the loop. Thus this loop takes $O(m^{0.5})$ time. The loop on line 6 populates the first column of L at the locations of each square sq in SQ. As $|SQ| \in O(m^{0.5})$, this loop takes $O(m^{0.5})$ time as well. The loops on lines 9, 10, and 11 iterate over each column from 2 to d, then over each square sq in SQ, then over each row from 1 to $m - sq$. This represents the majority of the time spent and is in $O(dm^{0.5}m) = O(dm^{1.5})$. The last operation executes in constant time. Thus the entire algorithm runs in $O(md) + O(m^{0.5}) + O(dm^{1.5}) + O(1) = O(m^{1.5}d)$ time.

Q3 - Job allocation

We present a solution to the problem specified below which reduces the problem to the Maximum Flow problem, and uses the Floyd-Fulkerson Algorithm to solve it.

Formal specification of problem

Instance:

Given: A set of n jobs, J, and a set of m machines, C; a list for each job j_i of the machines capable of performing the job, $L_i = \{c_a, c_b, \ldots\}$; an overhead $1 \leq k \leq n$, the maximum number of jobs that a machine can perform.

Solution Space:

$True$ or $False$, which answers the question: Is there an assignment (a matching of each job to a machine capable of running it) with overhead at most k?

Constraints:

If it is possible to match each job j_i to a machine on its list L_i, such that there are not more than k jobs matched to any machine, the solution must return $True$. Otherwise, it must return $False$.

Objective:

Determine whether there exists such an assignment efficiently.

Definitions:

In addition to the variables defined in the problem specification, we define the following variables used in the algorithm:

- $s \in V$ is the start node of the maximum flow problem.
- $t \in V$ is the end node of the maximum flow problem

Algorithm and Analysis:

Algorithm:

```
Create graph G = (V, E);
Initialize V ←, E ←;
V ← {s, t};
for each c_i ∈ C do
    Create a node m_i that represents machine c_i;
    V ← V ∪ {m_i};
    Create a directed edge e = (m_i, t) with capacity c(e) = k;
    E ← E ∪ {e};
end
for each j_i ∈ J do
    Create a node n_i that represents job j_i;
    V ← V ∪ {n_i};
    Create a directed edge e = (s, n_i) with capacity c(e) = 1;
    E ← E ∪ {e};
    for each c_j ∈ L_i do
        Create a directed edge e = (n_i, m_j) with capacity c(e) = 1, where m_j is the node that
        represents machine c_j;
        E ← E ∪ {e};
    end
end
f = Floyd-Fulkerson(G, s, t);
if F(f) = n then
    return Yes;
end
else
    return No;
end
```

We solve the job allocation problem by reducing it to the maximum flow problem. We construct a graph where the start node s has an edge of capacity 1 to each of the jobs. Each job has an edge of capacity 1 to each of the machines that can run it. Finally, each machine has an edge to the end node t with capacity equal to the overhead k, which prevents more than k jobs from being assigned to any machine. Next, we find the maximum flow from s to t, where a flow from a job's node to a machine's node represents assigning the job to that machine. If the maximum flow is equal to the number of jobs, n, that means that each job was assigned without violating the overhead restriction. If not, then it is not possible to assign each job to a machine with the overhead restriction.

Correctness:

Proof. A maximum flow from s to t represents an assignment of each job to a machine that respects the overhead restriction of no more than k jobs assigned to a single machine. If the maximum flow utilizes the edge from s to every job, then the maximum flow will be n, the number of jobs. The maximum flow cannot exceed n because there are exactly n edges leaving s, and each has capacity 1. No job can be assigned to multiple machines, because the conservation property of a flow requires that the total flow leaving each node (besides s and t) equal the total flow entering it, and every job has exactly one edge entering it with capacity 1. No machine can be assigned more than k jobs, because each job consumes 1 capacity, and by the same conservation and capacity argument each machine has only one edge leaving it, with capacity k. Thus, the maximum flow from s to t is n if and only if there is an assignment with overhead at most k.

The maximum flow from s to t in graph G is returned by `Floyd-Fulkerson(`G, s, t`)`. If `Floyd-Fulkerson(`G, s, t`)` returns a maximum flow of size n, then there is an assignment where $\forall e = (n_i, m_j) \in E, n_i \neq s \wedge m_j \neq t \wedge F(e) = 1 \Rightarrow$ assign job i to machine j, which satisfies the overhead constraint. If the size of the maximum flow is less than n, then there does not exist an assignment of all jobs that satisfies the constraints. If such an assignment existed, then we could construct a flow where for each assignment of job i to machine j, $F((n_i, m_j)) = 1$, $F((s, n_i)) = 1$, and $F((m_j, t)) = \sum_{e=(p,m_j)} F((p, m_j)) \leq k$. For all other edges $e \in E$, $F(e) = 0$. If the assignment assigns all jobs to a machine that can run it, then the size of this flow is n. If such a flow existed, `Floyd-Fulkerson()` would have found it, so this is a condtradiction. Therefore, our algorithm correctly reports whether an assignment of every job to a machine that satisfies the contraints is possible.

□

Time Analysis:

Our algorithm consists of two steps: constructing graph G, and calling `Floyd-Fulkerson()`. Construction of the graph involves creating $|V| = n + m + 2$ nodes and $|E| = n + m + |L|$ edges. L is the list of which machines can perform each job. Thus, the construction of the graph takes $O(|V| + |E|) = O(n + m + |L|)$. The runtime of `Floyd-Fulkerson()` is $O(|F_{OPT}||E|)$. $|F_{OPT}| \leq n$ because the size of the maximum flow is bounded by the sum of the capacities of all of the edges leaving s, and there are n edges leaving s, each with capacity 1. Thus, the `Floyd-Fulkerson()` call takes $O(n(n + m + |L|))$. Therefore, the whole algorithm takes $O(n+m+|L|)+O(n(n+m+|L|)) = O(n(n+m+|L|))$. In the worst case, $|L| = n \times m$, where every job can be run on every machine. Then, the algorithm's overall performance is $O(n^2 m)$.

In the case where $m = n/\log n$, the algorithm's runtime is $O(n^3/\log n)$.

Q4 - Finding Bottleneck edges

We present a network-flow solution to the bottleneck edges problem described below.

Part a:

Formal specification of problem

Instance:

A directed graph G with two distinguished nodes s and t.

Solution Space:

An integer number of pair-wise edge-disjoint s-t paths in G.

Constraints:

Each path P must be pair-wise edge-disjoint.

Objective:

Find the maximum path number k of G (maximum number of pair-wise edge-disjoint s-t paths).

Algorithm and Analysis:

The template of this algorithm and analysis is taken from pages 374-376 of the class text.

Algorithm:

Our goal is to be able to use the Ford-Fulkerson algorithm on the graph so that we only need to calculate the maximum s-t flow in order to determine the maximum path number k of G.

Consider the graph $G = (V, E)$ with two distinguished nodes s and t. We define a valid flow network on G, as defined on pages 338-339 of the text, as follows: s is the source, t is the sink, and the capacity of each edge in G is 1, i.e. $\forall e \in G, c(e) = 1$. We want to set the capacity of each edge in G to 1 so that in the case where a node n had more incoming edges than outgoing edges, the conservation property would ensure that the incoming edges that are not matched with an outgoing edge are not used in a path from s to t. If any of the unmatched incoming edges were part of a path from s to t, then that implies that a path leaving n would have a flow of more than 1 which contradicts our assumption about the flow of each edge being 1 and thus the paths would not be pair-wise edge-disjoint because the outgoing edge would then be carrying the unit flows from more than one incoming path.

We first prove the following property about a graph with a path number of k, moving us one step closer to our goal.

Lemma 3. *If there are k pair-wise edge-disjoint s-t paths in a directed graph G, then the value of the maximum s-t flow F in G is at least k.*

Proof of Lemma 3. Suppose G has k pair-wise edge-disjoint s-t paths with a flow of 1. This means each edge e on each path p has a flow of 1, i.e. $f(e) = 1$. Further, because of the conservation property of flows and because each edge has integer flow values, this implies that all other edges e' have a flow of 0, i.e. $f(e) = 0$. Thus since all edges are disjoint, have a flow of 1, and there are k edges from the k paths entering t, the maximum flow F of the graph is k. □

Now that we have proved that if there are k pair-wise edge-disjoint s-t paths in a directed graph G, then the value of the maximum s-t flow F in G is k, we must prove the other direction so that we can use the Ford-Fulkerson algorithm to determine the maximum path number k. In doing so, we will prove the following algorithm constructs k pair-wise edge-disjoint s-t paths when given a graph G constrained as above and with maximum flow F by decomposing the flow pair-wise edge-disjoint s-t paths. The following algorithm is taken from page 375 of the text.

```
1  MaxDisjointPaths(F on G):
2  begin
3      P ← ∅;
4      while there is an edge e ∈ F with f(e) = 1 do
5          Start at node s;
6          Continue constructing a path until one of the following happens;
7              1) t is reached, then:;
8                  Change flows of all edges along s-t path p to 0;
9                  F ← F − e ∈ p;
10                 P ← p;
11             2) a node v in the path is visited a second time, then:;
12                 Change all edges along cycle c between first and second occurrence of v to 0;
13                 F ← F − e ∈ c;
14     end
15     return Set P of pair-wise edge-disjoint s-t paths.;
16 end
```

We next prove the following property about a graph with a maximum flow of value at least k, allowing us to prove the optimality of using the Ford-Fulkerson algorithm to find the maximum path number.

Lemma 4. *If the value of the maximum s-t flow F in G is at least k, then there are k pair-wise edge-disjoint s-t paths in a directed graph G.*

Proof of Lemma 4. From (7.14) in the text, we know that there is a maximum flow F with integer values. Since all edges have capacity $c(e) = 1$ and the flow is integer valued, we know that each edge that is part of the maximum flow F has flow $f(e) = 1$. Thus we actually can prove the lemma by proving the following statement:

If F is a 0-1 valued flow of value v, then the set of edges with flow value $f(e) = 1$ contains a set of v pair-wise edge-disjoint s-t paths.
We will prove this statement by induction on the number of edges D in F.
Base case: If the maximum flow value $v = 0$, there is no flow in the graph and there are no s-t paths.
Inductive case: Since $v > 0$, s must have at least one outgoing edge to u with $f((s, u)) = 1$. This corresponds to line 5 in the algorithm above. We then continue to construct a path along edges with $f(e) = 1$ because we know by the conservation property that u has an outgoing edge and thus there is an edge (u,v). This continues until we either reach our sink t or we reach a node v that we have already visited, due to a cycle in the graph. We consider the two cases separately:

a) We reach t. In this case, we have an s-t path p. Following line 8 of the algorithm, we set all edges along p to 0 which will reduce our total maximum flow by 1 to $v - 1$ in addition to the total number of edges that can carry flow. We call this new flow F' and it is produced by line 9. Thus, applying our induction hypothesis to F', we now have $v - 1$ pair-wise edge-disjoint s-t paths. Finally, on line 10, for use in the next problem, we save p in the set of pair-wise edge-disjoint s-t paths P. Taking the union of the F' and P, we see that we have v paths as stated above.

b) We reach v for the second time. In this case, we have a cycle c. Following line 12 of the algorithm, we set all edges along c to 0 which will also reduce our total number of edges that can carry flow, but will not reduce the maximum flow. We call this new flow F' and it is produced by line 13. Thus, applying our induction hypothesis to F', we have v paths as stated above.

Thus, if the value of the maximum s-t flow F in G is at least k, then there are k pair-wise edge-disjoint s-t paths in a directed graph G. □

Since we have proved both directions, we can say the following:
There are k pair-wise edge-disjoint s-t paths in a directed graph G from s to t if and only if the value of the maximum value of an s-t flow in G is at least k.

Thus, since we have proved that the maximum flow value of G is equal to the path number of G, we can use the Ford-Fulkerson maximum-flow algorithm on page 344 of the text to find the path number of G (maximum set of pair-wise edge-disjoint s-t paths).

Time Analysis:

The Ford-Fulkerson algorithm is proved to be bounded by $O(mC)$ in (7.5) in the text. C is the capacity out of s, the source. In this case, C is bounded by $O(n)$ because $c(e) = 1$ for all edges leaving s and there are at most $|V| - 1 = n - 1$ edges out of s, since there is no edge from s to itself. Thus, substituing n for C, the runtime to find the path number is bounded by $O(mn)$.

The `MaxDisjointPaths()` algorithm defined above is used for the next problem, so we analyze the time here. Since only the edges in the pair-wise edge-disjoint s-t paths are part of F and since there can be at most $n - 1$ edge-disjoint paths from s to t due to the constraint that each must use a different edge out of s, the while loop on line 4 takes $O(n)$ time. Inside the while loop, the time is dominated by the search from s-t which can be made as efficient as a depth-first or breadth-first search which will take $O(m + n)$ time as claimed by (3.11) and (3.13) in the text. But since all nodes have at least one incident edge, $m \geq \frac{n}{2}$, so we can say that it is bounded by $O(m)$. Thus the run-time of the algorithm as a whole is $O(mn)$.

Part b:

Formal specification of problem

Instance:

The directed graph $G = (V, E)$ with two distinguished nodes s and t and $\forall e \in G, c(e) = 1$. We also need the set P, built from G by `MaxDisjointPaths()` containing the set of all k pair-wise edge-disjoint s-t paths.

Solution Space:

A set of edges B.

Constraints:

An edge is a bottleneck edge if removing it from G decreases the path number.

Objective:

Find all the bottleneck edges $b \in P$.

Algorithm and Analysis:

Algorithm:

The following is the algorithm to find all bottleneck edges in a graph. `MaxPath(`G`)` is the Ford-Fulkerson algorithm as described on page 344 of the text. `FindSimplePath(`u, v, G`)` finds a simple path between u and v in G using either breadth first or depth first search as discussed on pages 78-94 in the text.

```
1  FindAllBottlenecks(G,P):
2  begin
3      F ← MaxPath(G), also keep G_R, the residual graph of G for future use.;
4      P ← MaxDisjointPaths(F);
5      ∀e ∈ E, c(e) = 1, f(e) = 0;
6      foreach path p ∈ P do
7          foreach edge e ∈ p do
8              G'_R ← G_R;
9              remove e from G'_R;
10             set all forward edges in G'_R along p to 1 and all backward edges in G'_R along p to 0;
11             if FindSimplePath(s, t, G'_R) == false then
12                 B ← B ∪ e;
13             end
14         end
15     end
16     return Set of bottleneck edges B;
17 end
```

Correctness:

We show that the algorithm will find all bottlenecks.

Lemma 5. *The algorithm* `FindAllBottlenecks(`G,P`)` *will find a set of B bottleneck edges in the graph G.*

Proof of Lemma 5. In the first step of the algorithm, we calculate the MaxPath on the graph using the Ford-Fulkerson algorithm, which returns a flow F over the graph G, in addition, we store its residual graph G_R for use by the algorithm. This provides input the the `MaxDisjointPaths()` algorithm discussed in Part a and outputs the set of pair-wise edge-disjoint s-t paths. The set of paths P are required in order to determine the current configuration of G.

At the beginning of the main loop, we only consider the edges e that are in the k pair-wise edge-disjoint s-t paths of G because any other edges not in the paths will obviously not reduce the path number if removed and thus can never be a bottleneck. The first step that we take is to copy the residual graph G_R to a temporary graph G'_R so that G_R is not modified each iteration. This guarantees that we are always starting from the solution's residual graph and only ever removing one edge from it at a time, meaning we can determine the effect of a single edge on the path number if removed. Next, the edge e is removed from the residual graph G'_R and thus implicitly from G so that parts 2 and 3 of the definition of residual graph continue to be satisfied. We do not need to actually update G because it is not used directly in determining bottleneck edges, only its residual graph with its augmenting edges. We only talk about updating the real solution G in this proof of correctness. Since our hypothetical G can now be considered to have one less edge, the path p from which e was chosen is now broken, reducing the number of paths in G by 1. Thus there are now $k-1$ paths in G and by our results in Part a, the flow is reduced to $k-1$ as well.

At this point, we want to determine if there is a redundant edge such that the path number (and thus flow) can be increased back to k. In order to maintain the conservation property in the graph, flows of each edge on the broken path p in G must be zeroed out. To maintain consistency with G'_R according to the residual graph definition, we must also set its corresponding forward edges to 1 and zero its corresponding backward edges along path p. A path finding algorithm `FindSimplePath` () is then used to determine if there is a simple path from s to t in G'_R, in the same manor as the Ford-Fulkerson algorithm. If there is an augmenting, simple path from s to t, then we know that a redundant edge was found, replacing the deleted one. Since we found an augmenting path, the path number of G can once again be increased to k and by definition, the edge e is not a bottleneck since it did not reduce the path number after being removed. If there is no simple path from s to t, then we know that edge e was critical to the k^th path from s to t and thus a bottleneck. In this case, edge e is added to the set of bottleneck edges. Thus, in each iteration, after deleting an edge, our hypothetical G has either k or $k-1$ paths which is sufficient to declare an edge redundant or a bottleneck, respectively, by the definition of a bottleneck edge. □

Lemma 6. *The set of bottleneck edges B return by* `FindAllBottlenecks(G)` *is optimal.*

Proof of Lemma 6. Assume the optimal set of bottlenecks O is larger than B, the set returned by `FindAllBottlenecks(G)` (i.e. $|O| > |B|$). Thus O contains at least one edge e' that B does not. Since all edges in the k pair-wise edge-disjoint paths were considered when constructing B, that means that the optimal algorithm determined that there were no other paths in G after e' was removed. But when `FindAllBottlenecks(G)` considered e', it did find a path in G after e' was removed. Since we have proved our algorithm correct, then the optimal algorithm must be incorrect which is a contradiction. Thus, the set bottleneck edges B returned by `FindAllBottlenecks(G)` is optimal. □

Time Analysis:

This algorithm is significantly more efficient than the brute force method of taking the original graph G, deleting an edge, and then running the Ford-Fulkerson algorithm on it for each edge in G. This would take $O(m^2C)$ which is slow.

The time analysis of `FindAllBottlenecks(G)` as described above can be broken down as follows. The calls to `MaxPath()` and `MaxDisjointPaths()` take $O(mn)$ time each as proved in Part a. The nested for loops on lines 6 and 7 simply iterate over all edges in P. We know that since the edges are all disjoint, there are at most $n-1$ edges in P, the set of all pair-wise edge-disjoint s-t paths. Thus, together they run in $O(n)$ iterations. In the body of the nested for loops, setting the edges in G'_R along p will take $O(m)$ time. The algorithm to find the simple path from s to t will take $O(m+n)$ time as claimed by (3.11) and (3.13) in the text. Since all nodes have at lest one incident edge, $m \geq \frac{n}{2}$. Thus, the search for a simple path takes $O(m)$. Thus, the nested for loops take asymptotically have $O(mn+mn) = O(mn)$ time. This gives a time complexity of $O(mn+mn+mn) = O(mn)$ for the algorithm as a whole.

Q5 - Implementation: Independent Set

We implemented a greedy algorithm to find approximate solutions to the Maximum Independent Set problem using the heuristic specified. We also implemented a backtracking algorithm to find the optimal solution, based on the third backtracking algorithm presented in class. Both algorithms were implemented in C. We ran a set of experiments using graphs with nodes of 64, 128, and 256 where each node in the graph has edges with probability 1/2, 1/4, and 1/8. The values from the experiments were averaged over 10 runs each. All experiments were run on a 2.2 GHz MacBook Pro. Our results are listed in the table below.

Table 1: Average sizes of Maximum Independent Set

n	64			128			256		
p	Greedy	BT	G/BT	Greedy	BT	G/BT	Greedy	BT	G/BT
1/2	8	8	**1**	9	9.9	0.91	9	11.7	0.77
1/4	12.2	14.9	0.82	15	18.3	0.82	18	-	-
1/8	19	22.2	0.86	26	-	-	34	-	-

Although we were able to run the greedy algorithm on values of n greater than 256, we were unable to do so with the backtracking algorithm. The backtracking algorithm took too long to complete. In fact, the table shows "-" values where the backtracking algorithm was unable to complete within a reasonable amount of time. From these results we see that the maximum ratio is 1 where $n = 64$ and $p = 1/2$ (in bold in the table). This is no doubt the result of random chance. We do see that as the number of nodes increases, the ratio decreases. The jump from 64 to 256 drops the ratio of $p = 1/2$ problems from 1 to 0.77. This is a considerable decrease given that the problem size has only increased by 4. By far the most interesting trend however, is that the more sparse the graph, the longer it takes the backtracking algorithm to find a solution. This is directly related to the fact that the backtracking algorithm prunes more nodes from the graph at each step if the node degrees are larger. This is evidenced by the fact that problems of size $n = 128$ with $p \leq 1/8$ and problems of size $n = 256$ with $p \leq 1/4$ simply could not be solved by the backtracking algorithm within a reasonable amount of time.