Probabilistic analysis

Gokul Subramanian,
Advised by: Prof. Sampath Kannan

March 24, 2012

Contents

1 Introduction

Algorithms are often compared either experimentally or by their theoretical worst case asymptotic behavior. The problem with the former method is that the results of testing on certain inputs might not imply the overall characteristics of the algorithm(s) over the entire input space. And the latter method is too pessimistic about the performance of algorithms.

A more realisitic approach to comparing algorithms is to analyse their average case asymptotic behavior. The definition of average (or mean or expected value) is based on an assumed probability distribution over the input space. This can be tricky and lead to a very complicated analysis. However, in some cases, a fairly reasonable probability distribution evidently stands out. This incomplete survey attemps to describe the average case analysis of a variety of algoritms.

We will cover commonly used algorithms from the two domains. (a) Sequence algorithms: Searching ordered arrays, Sorting, Selection and Hashing, and (b) Graph algorithms: Minimum Spanning Trees, Shortest paths and Bipartite matching. For each problem, we will give a short description of the problem and then proceed to describe the various algorithms and their corresponding analyses for that problem. All these algorithms are used both as an end in themselves, and as subroutines in other more complicated algorithms. Karp [9] presents an extensive bibliography of probabilistic analyses.

2 Searching ordered arrays

Suppose we have an array X of $|X| = n$ objects. The indices in X start at 1 and end at n. Each object has a key that comes from some set R. The objects are arranged in an ascending order (defined on R). Let X_r denote the key of the object at index r. The inputs to the algorithm are the array X and a search key k. The output has two possibilities. If an object with key k is present in X, then output must be the index of that object in X. If no object with

key k exists, then output must be -1. We consider two algorithms for this problem: Binary search and Interpolation Search, and analyse their behavior.

2.1 Binary search

Binary search works as follows. At step j, the algorithm maintains a state $S_j = (L_j, U_j)$, where L_j connotes a lower/left index and U_j connotes an upper/right index. If $L_j \geq U_j$, the algorithm outputs -1 i.e. key not found. Initially, $S_1 = (0, n+1)$. At step j, the algorithm computes the middle index $M_j = \lfloor (L_j + U_j)/2 \rfloor$, and compares X_{M_j} with the search key k. If these are equal, then it outputs M_j. If $X_{M_j} > k$, then the new state S_{j+1} is defined as $S_{j+1} = (L_j, M_j)$. Else if $X_{M_j} < k$, then the new state S_{j+1} is defined as $S_{j+1} = (M_j, U_j)$. In the true sense of the term, this implementation resembles a ternary search because at each step, a 3-way decision is made. However, this small detail does not affect the overall analysis.

Roughly speaking binary search goes to the middle of the array, and checks if the key is present. If not then it discards one half of the array because of the assumed order on the elements of the array. Thus, the algorithm runs in $O(logn)$. If we assume that the objects in X have distinct keys, and that the search key is uniformly at random chosen to be one of the keys in X, then the average case runtime is given as follows.

$$T(n) = \frac{1}{n}(1) + \frac{2}{n}(2) + \frac{2^2}{n}(3) + ...\frac{2^{log_2 n - 1}}{n}(log_2 n)$$

$$\therefore T(n) = log_2 n - (1 - \frac{1}{n}) \sim \Theta(logn)$$

This is because there is one search key that takes one unit of time. There are two search keys that take 2 units of time, 4 search keys that take 3 units of time and so forth. And the $1/n$ factor is a result of the uniform probability distribution assumption.

2.2 Interpolation search

Interpolation search assumes that the objects of X have keys in a certain range $[a, b]$. At each step j, the algorithm maintains a state $S_j = (L_j, U_j, X_{L_j}, X_{U_j})$, where both the left and right ends L_j and U_j of the remaining array are assumed to have been probed. The initial state is $S_1 = (0, n+1, a, b)$. The following analysis is from Perl [1].

At step j, if $L_j \geq U_j$, then the algorithm outputs -1. Else it computes the following two values: $P_j = \frac{k - X_{L_j}}{X_{U_j} - X_{L_j}}$, $N_j = U_j - L_j - 1$, and then the next probe index $K_j = \lceil L_j + P_j N_j \rceil$. The algorithm then compares X_{K_j} and k. If these are equal, then it outputs K_j. If $X_{K_j} > k$, then the new state S_{j+1} is defined as $S_{j+1} = (L_j, K_j, X_{L_j}, X_{K_j})$. Else if $X_{M_j} < k$, then the new state S_{j+1} is defined as $S_{j+1} = (K_j, U_j, X_{K_j}, X_{U_j})$.

Consider the very rare array X such that $X_1 = n/2$ and $\forall i > 1, X_i = (n/2+1)$. Say the search key is $k = n/2$ and that the initial state is $S_1 = (0, n+1, 0, n)$. The first probe will be $K_1 = \lceil 0 + \frac{n/2-0}{n-0}(n+1-1) \rceil = n/2$. Thereafter, the search will shift to the new state $S_2 = (0, n/2, 0, n/2+1)$. So, the next probe will be at $K_2 = \lceil 0 + \frac{n/2-0}{n/2+1-0}(n/2-1) \rceil = n/2 - 1$. In this way, interpolation search will probe all indices from $n/2$ to 1, and therefore has a runtime of $O(n)$.

For the average case analysis, let us assume that the keys in the index range $F_j = L_j + 1....U_j - 1$ are uniformly distributed in $[X_{L_j}, X_{U_j}]$. This is a fairly restrictive assumption but nevertheless allows some interesting analysis. We will also assume that the key k is present in the array X. Let K^* denote the actual index of k in F_j. Then, $K_j = E(K^* | S_1, S_2, ...S_j)$.

Proof: K^* is a random variable, which can be expressed as follows. $K^* = \lceil L_j + I_j \rceil$, where I_j is a binomially distributed random variable having expectation $N_j P_j$ and variance $N_j P_j (1 - P_j)$. This is because P_j represents the probability that a randomly chosen key in F_j is less than or equal to k and $N_j = |F_j|$. q.e.d.

Now, let $D_j = |K_{j+1} - K_j|$ denote the change in our probe index from step j to step $j+1$. Thus, $D_j = |E(K^* | S_1, S_2, ...S_{j+1}) - E(K_j | S_1, S_2, ...S_{j+1})| = |E(K^* - K_j | S_1, S_2, ...S_{j+1})|$. Thus, D_j can be thought of either as a measure of the convergence of the series K or as the expected value of the error at step j. Further, since K_j is equal to either L_{j+1} or U_{j+1}, D_j is respectively equal to either $N_{j+1} P_{j+1}$ or $N_{j+1}(1 - P_{j+1})$.

We claim that $\forall j > 1, E(D_j^2 | S_1, S_2, ...S_j) \leq D_{j-1}$.

Proof:

$E(D_j^2 | S_1, S_2, ...S_j)$

$= E(E(K^* - K_j|S_1, S_2, ...S_{j+1})^2|S_1, S_2, ...S_j)$

$\leq E(E((K^* - K_j)^2|S_1, S_2, ...S_{j+1})|S_1, S_2, ...S_j)$

$= E((K^* - K_j)^2|S_1, S_2, ...S_j)$

$= N_j P_j(1 - P_j) \leq D_{j-1}$ q.e.d.

Now, we prove two important results about the algorithm that conclude the average case analysis of interpolation search. The first result is that the average error after j steps is less than $n^{2^{-j}}$. And as a corollary that after $log_2 log_2 n$ steps, the average error is less than 2.

Proof: Note that $E(D_1|S_1)^2 \leq E(D_1^2|S_1) \leq N_1 P_1(1 - P_1) = nP_1(1 - P_1) \leq n/4$. Thus, $E(D_1|S_1) < \sqrt{n}$.

Also, $E(D_j|S_1)^2 \leq E(D_j^2|S_1)$

$= E(E(D_j^2|S_1, S_2, ...S_j)|S_1) \leq E(D_{j-1}|S_1)$

Thus, by repeating this reduction process, we get $E(D_j|S_1)^{2^{j-1}} \leq E(D_1|S_1) < \sqrt{n}$. By moving the exponentiation from the LHS to the RHS, we get $E(D_j|S_1) < n^{2^{-j}}$. q.e.d.

Now, for the next result. The average number of steps in interpolation search is upper bounded by $log_2 log_2 n$.

Proof: Consider our previously derived inequality. $E(D_j^2|S_1, S_2, ...S_j) \leq D_{j-1}$. Take logarithms to the base 2, define $Z_j = log_2 D_j$ and rewrite the inequality as $E(2^j Z_j|S_1, S_2, ...S_j) \leq 2^{j-1} Z_{j-1}$. Thus, the sequence $2^j Z_j$ is a supermartingale relative to the sequence S_j. Let T be the first j for which $Z_j \leq 1$. Lets assume that $Z_j = 1$ indeed for some $j = T$. Then, it is clear that since T is a stopping time, the Optional sampling theorem dictates that $E(2^T Z_T) = E(2^T) \leq E(2^1 Z_1) = E(2log_2 D_1) = E(log_2 D_1^2) \leq log_2(E(D_1^2)) < log_2 n$. So, $E(2^T) < log_2 n$. And by taking log to the base 2, we get $E(log_2 2^T) \leq log_2 E(2^T) < log_2 log_2 n \Rightarrow E(T) < log_2 log_2 n$. We need to patch up one last thing. The sequence Z_j might never take on a value of 1, because it is a discrete sequence. To amend this, we make $Z(t)$ a continuous function which derives from Z_j by linear interpolation. Now, we can can set T to be the first t such that $Z(t) = 1$. It can be verified that the supermartingale property still holds for this new function $Z(t)$.

Let us try to interpret the result $E(T) < log_2 log_2 n$. T measures the number of steps it takes for $Z_j = 1$ i.e. $log_2 D_j = 1$ i.e. $D_j = 2$ i.e. for the expected error to go to two. This is precisely the running time of the algorithm. Thus, interpolation search takes an average of $O(loglogn)$ time. We can also prove that the running time will not deviate way above the average. Using markov's inequality, we get $Pr(T \geq log_2 \beta) = Pr(2^T \geq \beta) \leq E(2^T)/\beta < log_2 n/\beta$. Set $\beta = (log_2 n)^{(1+a)}$. Then we get, $Pr(T \geq (1 + a) log_2 log_2 n) < (log_2 n)^{-a}$.

3 Sorting

Suppose we have an array X of $|X| = n$ objects. The indices in X start at 1 and end at n. Each object has a key that comes from some set R. The objects are arranged in a some order (defined on R). Let X_r denote the key of object at index r.

Any candidate sorting algorithm must output the array X' which satisfies two properties: (a) The multiset M consisting only and all of the objects in X should be same as the multiset M' consisting only and all of the objects in X'; and (b) The objects in X' should be arranged in ascending order of key.

We will consider the following sorting algorithms: quick sort, merge sort and insertion sort.

3.1 Quick sort

Quick sort works as follows. The algorithm first runs a partitioning subroutine. This subroutine chooses a pivot key X_p, and then by a single pass through the array, rearranges it such that objects with keys smaller than or equal to X_p come first, then comes the pivot object with key X_p and then come objects with keys larger than X_p. Call this array X'', and the rank of X_p in X as k. Then, quick sort recursively invokes itself on the two partitions $1...k-1$ and $k+1....n$ of X''. The base case is when there is one object in a partition, and in this case, the partition is already sorted.

The worst case for quick sort occurs when at each step, the chosen pivot has rank 2. To see this, call the time take by quick sort on an input of size n under this partitioning run as $T(n)$. And notice that each such (lousy) partitioning step will produce two subarrays, one of size 1 and the other of size $n-2$, and while the size 1 subarray needs no sorting,

the size $n-2$ subarray needs $T(n-2)$. Thus, the recurrence is $T(n) = cn + T(n-2)$, which leads to $T(n) \sim \Theta(n^2)$. Thus, quick sort has a runtime of $O(n^2)$.

Despite this poor worst case performance, quick sort is a very popular sorting algorithm. This is because of the fact that the worst case partitioning run happens only very rarely. Thus, it might be insightful to understand the average runtime of quick sort. To do this, let us assume that all orderings of the input are equally probable. Intuitively, since a deterministic partition scheme can pick any element in array X as pivot, the expected rank of the pivot is $n/2$. Thus we have $T(n) = n - 1 + 2T(n/2) \Rightarrow T(n) \sim \Theta(nlogn)$. The following rigorous analysis yields the same result.

$$T(n) = n - 1 + \frac{1}{n}\sum_{i=1}^{n-1}(T(i) + T(n-1-i))$$
$$\therefore T(n) = n - 1 + \frac{2}{n}\sum_{i=0}^{n-1} T(i)$$

$$\therefore nT(n) - (n-1)T(n-1) =$$
$$n(n-1) - (n-1)(n-2) + 2T(n-1)$$
$$\therefore \frac{T(n)}{n+1} = 2\frac{(n-1)}{n(n+1)} + \frac{T(n-1)}{n}$$
$$\therefore \frac{T(n)}{n+1} = 2(\frac{2}{n+1} - \frac{1}{n}) + \frac{T(n-1)}{n}$$

$$\therefore \frac{T(n)}{n+1} = 2\sum_{i=1}^{n}(\frac{2}{i+1} - \frac{1}{i})$$
$$\therefore \frac{T(n)}{n+1} = 2(2H_{n+1} - 1 - H_n)$$
$$\therefore \frac{T(n)}{n+1} < 2(H_{n+1} - 1)$$

$$\therefore T(n) \sim O(nH_n) \sim O(nlogn)$$

3.2 Merge sort

Merge sort also uses a divide and conquer strategy like quick sort. It sorts the two halves of array X and then merges the two sorted halves in $\Theta(n)$ time. Thus, the running time of merge sort is $O(nlogn)$. The worst case asymptotic behavior of merge sort is better than that of quick sort, but in practice these algorithms must be compared on a set of realistic inputs.

3.3 Insertion sort

Insertion sort is an online adaptive in-place sort algorithm. The algorithm works as follows. At every step, the algorithm maintains an array A of objects ordered by the object keys. Every time a new object is received, it is stored in A at the correct index according to the sort order. Clearly, the worst case of $O(n^2)$ happens when the incoming sequence is in descending order. Knuth [5] specifies the exact running time as $O(n+I)$, where I is the total number of inversions i.e. incorrectly ordered pairs in the array. Thus, if we know beforehand that $I \sim O(n)$, then insertion sort runs in linear time.

To compute the average number of inversions, we notice that if an array has I inversions, then its reversed version (also called transpose) has exactly $\frac{n(n-1)}{2} - I$ inversions. Thus, for a random ordering of the array, we expect that the array and its transpose will have the same number of inversions i.e. $n(n-1)/4$; insertion sort takes $O(n^2)$ on average.

4 Selection

Suppose we have an array X of $|X| = n$ objects. The indices in X start at 1 and end at n. Each object has a key that comes from some set R. The objects are arranged in a some order (defined on R). Let X_r denote the key of object at index r. We are also given a positive integer $k \in [1, n]$. The selection algorithm must output the object (or its key) in X that has rank k.

One way to select the rank k object from X is to sort X and then pick the object at index k. But this requires $O(nlogn)$ time. While this might be a good idea if $\Omega(logn)$ selections need to be made from the same array, it is very expensive when only a few selections are required. Indeed, we can run selection in $O(n)$ time. The quick select algorithm described below gives an expected $O(n)$ runtime. There is a deterministic worst case $O(n)$ time algorithm called the Median of Medians algorithm that has bigger hidden constants and is thus less practical for typical values of n.

4.1 Quick select

Quick select works as follows. The algorithm randomly chooses a pivot X_p, and then by a single pass through X, rearranges it such that objects with keys smaller than or equal to X_p come first, then comes the pivot object X_p and then come objects with keys larger than X_p. Call this array X'' and the rank of X_p in X as r. If $k = r$, then the rank k object X_p has been found, and quick select will output that. Else if $k < r$, quick select recursively invokes itself on the 1...r-1 partition of X'' with a rank k object to be selected. Else if $k > r$, then quick select recursively invokes itself on the r+1...n partition of X'' with a rank $k - r$ object to be selected.

In the worst case, this algorithm runs in $O(n^2)$. But, the average case is $O(n)$ as shown in Balcan [3]. Call the expected time taken on an instance of size n, k as $T(n, k)$. And let $T(n) = \max_k T(n, k)$. Then, the following recurrence summarizes $T(n)$.

$$T(n) < cn + Avg(T(n/2), T(n/2 + 1), ...T(n - 1))$$

The cn term is due to the linear time spent in initially rearranging the array. The second term is averaged over all possible sizes of X''. Indeed we are being pessimistic here because X'' could also be of size $s < n/2$, but in that case we replace s by $n - s$ in the analysis knowing in advance that $T(s) < T(n - s)$.

We can solve this recursion by the trial and error method. Let us assume that $T(i) < bi$ for all $i < n$. Then, $T(n) < cn+Avg(bn/2, b(n/2+1), ...b(n-1)) = cn + (b/(n/2))(n^2/4 + n^2/8 - n/4) = cn + b(3n/4 - 1/2) < cn + 3bn/4$. By choosing $b > 4c$, we get $T(n) < bn$. Thus, $T(n) \sim O(n)$ by induction.

5 Hash tables

In this section, we will discuss some fundamental results taken from Aspnes [2] for one of the most useful data structures, the hash table. The most primitive hash table utilizes a deterministic hash function $h : U \to M$, where U, the domain is called universe and M, the range is equal to the set of integers $[1, m]$.

The idea here is that an object with key $u \in U$ is stored in an array A at index $h(u)$. If two distinct keys $x, y \in U$ hash onto the same index i.e. $h(x) = h(y)$, we say that there is a collision. In practical implementations of hash tables, collisions are most often resolved by chaining. In this method, each index in the array A maintains a pointer to a linked list of keys that hash onto that index.

Clearly, if $|U| > m$, then by the pigeonhole principle, there must be at least one collision. More generally, if $n \geq 1$ and $|U| = (n - 1)m + 1$, then there is a set $S \subseteq U$ of n keys, which all mutually collide; otherwise $|U| \leq (n-1)m$, a contradiction. Intuitively, this means that for large $|U|$, and a given hash function h, an adversary can always find a large set $S \subseteq U$, all of whose keys collide. This leads to the idea of universal hash families.

5.1 Universal hash family

A family H of hash functions is a set of hash functions. H is said to be 2-universal if for any $x, y \in U$, and a randomly chosen $h \in H$, $Pr(h(x) = h(y) \wedge x \neq y) \leq 1/m$. Let us denote $\delta(x, y, h)$ to be 1 if $h(x) = h(y)$ and 0 otherwise. Lets also define $\delta(x, y, H)$ as $\sum_{h \in H} \delta(x, y, h)$. Thus, if H is 2-universal, $\forall x, y : \delta(x, y, H) \leq |H|/m$.

While we might get greedy and ask ourselves if we could do much better than with universal hash families, the following theorem puts a full stop to our hopes: For any family H of hash functions, there exists a pair of distinct keys $x, y \in U$ such that $\delta(x, y, H) \geq \frac{|H|}{m}(1 - \frac{m-1}{|U|-1})$.

Proof: The number of collisions for a particular index $z \in M$ is given by $\delta(h^{-1}(z), h^{-1}(z), h) = |h^{-1}(z)|(|h^{-1}(z)| - 1)$. Here we have abused notation by using $\delta(X, Y, h)$ to mean $\sum_{x \in X, y \in Y} \delta(x, y, h)$.

Summing over all $z \in M$, we get $\delta(U, U, h) = \sum_{z \in M} |h^{-1}(z)|(|h^{-1}(z)| - 1)$.

Constraining $\sum_{z \in M} h^{-1}(z) = |U|$, we can minimize the RHS by using Lagrange multipliers. This is equivalent to minimizing $\sum_i x_i^2$ constrained by $\sum_i x_i = c$ for which the solution is $x_1 = x_2 = ...$ By analogy, our minimizing $|h^{-1}(z)|$ is equal to $|U|/m$. Thus, we have $\delta(U, U, h) \geq \frac{|U|}{m}(|U| - m)$. Summing over all h, we get $\delta(U, U, H) \geq \frac{|H|}{m}|U|(|U|-m)$. Since there must be at least one number in an array of numbers that is greater than or equal to the average, for some distinct $x, y, \in U$, $\delta(x, y, H) \geq \frac{|H|}{m} \frac{|U|(|U|-m)}{|U|(|U|-1)} = \frac{|H|}{m}(1 - \frac{m-1}{|U|-1})$. q.e.d.

This result has a strong significance. It implies that for $|U| >> m$, there will be some distinct $x, y \in U$ such that $\delta(x, y, H) \to |H|/m$. Thus, the 2-universal guarantee is asymptotically the most optimal we can expect.

A 2-universal family of hash functions enables the hash table to have really good search, delete and update times. Suppose collisions are implemented by chaining. Suppose the hash function h is chosen uniformly at random from H when the hash table was initialized. Note that computing the chosen hash function on any key takes constant time in the size m of the array A, and the number of elements n in the array at any given time. Inserting a new key into the array the takes constant time, because the new key needs to be merely inserted at the head of the linked list at the computed hash index. This is under the assumption that multiple copies of the key are allowed. If this assumption is not valid for the problem at hand, then inserting can take as much time as searching and deleting. Searching and deleting both require a full traversal of the linked list at the computed hash index. Now, with a 2-universal has family, we know that for any search key x, the expected number of collisions with x is upper bounded by $(n-1)/m$. Thus, the length of the linked list at the index $h(x)$ is $O(n/m)$, and therefore search and delete both take $O(1 + n/m)$ time.

Below, we consider two 2-universal hash families, one for hashing integers and the other for hashing fixed length vectors.

5.1.1 Hashing integers

Consider the universe of keys $U = \{x | x \in \mathbb{Z}\}$. Consider the family H of hash functions defined as $h(x) = ((ax + b) \mod p) \mod L$, where $a \in \mathbb{Z}_p - 0$, $b \in \mathbb{Z}_p$, and p is a prime $\geq L$. a and b are chosen uniformly at random from their respective domains while initializing the hash table. We will use an array A of size L whose indices are numbered $0, 1, ...L-1$, so that the hash computations produce a valid array index.

We prove that H as defined above is 2-universal. Consider two distinct keys $x, y \in U$. For $h(x) = h(y)$, we must have $ax + b \equiv (ay + b + iL) \mod p$ i.e. $a(x - y) \equiv iL \mod p$. Since x and y are distinct, and this modulo prime arithmetic, and $a \in \mathbb{Z}_p - 0$, we get $a = \frac{iL}{x-y} \mod p$. Now, i can take at most $\lceil p/L \rceil$ values. We discount the case when $i = 0$ because for distinct x, y it is not possible that $ax + b \equiv (ay + b) \mod p$. Thus, effectively, there are $\lceil p/L \rceil - 1$ values of i. And a can take on $p - 1$ values. Thus, the collision probability is $(\lceil \frac{p}{L} \rceil - 1)\frac{1}{p-1} \leq 1/L$, as required for a universal hash family.

5.1.2 Hashing vectors

Consider the universe of keys $U = \{(x_1, x_2, ...x_k) | \forall i \in [1, k], x_i \in [0, L-1]\}$ for some fixed positive integer k. We will require that L be a prime number.

Consider the family H of hash functions defined as $h(x_1, x_2, ...x_k) = (\sum_{i=1}^{k} r_i x_i) \mod L$, where $\forall i \in [1, k], r_i$ is an integer chosen uniformly at random from $[0, L-1]$. We will use an array A of size L whose indices are numbered $0, 1, ...L-1$, so that the hash computations produce a valid array index.

We prove that H as defined above is 2-universal. Consider two distinct keys $x, y \in U$. It must be true that $x_i \neq y_i$ for some $i \in [1, k]$. Call $\sum_{j \neq i} r_j x_j$ as $h'(x)$ and likewise for $h'(y)$. Thus, if $h(x) = h(y)$, then $h'(x) + r_i x_i \equiv (h'(y) + r_i y_i) \mod L$. Equivalently, $r_i(x_i - y_i) \equiv (h'(y) - h'(x)) \mod L$. Since L is a prime, $r_i \equiv \frac{h'(y)-h'(x)}{x_i - y_i} \mod L$. Since $r_i \in [0, L-1]$, $r_i = \frac{h'(y)-h'(x)}{x_i - y_i} \mod L$. This gives us one unique value of r_i which causes a collision. Since r_i is chosen uniformly at random from among L different values, the probability of collision is $1/L$, as required for a universal hash family.

5.2 Perfect hash function

A perfect hash function $h : U \to M$ for a given set $S \subseteq U$ is one that is injective on S. i.e. for distinct $x, y \in S$, $h(x) \neq h(y)$. Thus, a perfect hash function can guarantee zero collisions and therefore $O(1)$ search time. The following theorem sheds light on how to find a perfect hash function: If H is 2-universal, $|S| = n$ and $n^2 \leq m$, then $\exists h \in H$ which is perfect for S.

Proof: For distinct $x, y \in S$, $\delta(x, y, H) \leq |H|/m$. So, $\delta(S, S, H) \leq n(n-1)|H|/m$. Because some number in an array of numbers must be less than or equal

to the average, we must have a $h \in H$ such that $\delta(S, S, h) \leq n(n-1)/m < n^2/m \leq 1$. But $\delta(S, S, h)$ is an integer, and must therefore be 0. q.e.d.

As a corollary, if $n^2 \leq \alpha m$, then the probability that a hash function $h \in H$ chosen uniformly at random causes a collision is equal to α. Thus, after $1/(1-\alpha)$ random picks, we should expect to get a perfect hash function. This idea is used in the following hashing scheme.

5.2.1 FKS hashing

In this technique, the static set S is first hashed onto an array of size $m = n$ the usual way, using a 2-universal hash family H . The bins with collision are then rehashed as follows. Consider a bin B_i with n_i keys. A new hash table is created with an array size of $m_i = \Theta(n_i^2)$. The keys are moved into this new hash table by a perfect hash function, chosen from a 2-universal H_{m_i} (the modified version of H, by replacing m with m_i). As we have already discussed, this might take a few attempts, but not too many. The chosen hash function h_i is also stored in the bin B_i. Any key search will now be first routed to the primary table, and then to the appropriate secondary table. Thus, the search operation takes $O(1)$ time. However, an even more interesting result is that the space consumption is $\Theta(n)$.

Proof: $\sum_i n_i^2 = \sum_i (n_i + n_i(n_i - 1)) = n + \delta(S, S, h)$. But H is 2-universal and therfore, $\delta(S, S, h) \leq n(n-1)/m = n - 1$, because $m = n$. Thus, the space consumption is $m + \sum_i n_i^2 = \Theta(n)$.

6 Minimum spanning trees

Consider an undirected graph $G = (V, E)$, where every edge $e \in E$ has some non-negative weight w_e. We will assume that G is connected. The weight of a subgraph H of G is defined as $\sum_{e \in H} w_e$. A spanning subgraph is one that is connected and has the same nodeset V as G. The problem is to find the minimum weight spanning subgraph. Evidently the minimum weight spanning subgraph H_{min} must be a tree, because otherwise, there would be a subset E' of edges in H_{min} that can be removed without disconnecting the graph. Thus, the problem boils down to finding the minimum (weight) spanning tree.

Two important properties described in Kleinberg [4], the cut property and the cycle property, underlie all algorithms that compute the minimum spanning tree.

Cut property: Let X be any subset of V. Then, the least weighted edge $e = (u, v)$ that connects X and $V - X$ must be a part of some minimum spanning tree.

Proof: Consider a spanning tree T' which does not contain e. Let us say that T' contains $e' = (u', v') \neq e$ connecting X and $V - X$. Since T is a spanning tree, it consists of a unique path $u \to u' \to v' \to v$. Modify T' by adding edge e and removing e'. The resulting graph is still a spanning tree, and has no lower weight. q.e.d.

Cycle property: For any cycle C in G, if the weight of an edge $e = (u, v)$ on C is greater than that of any other edge on C, then e cannot belong to any minimum spanning tree.

Proof: Consider a spanning tree T' that consists of such an edge e. The two nodesets X and $V - X$ that are connected by e must share the nodes of the cycle C. So, there must be some edge $e' = (u', v')$ on C such that $u' \in X$ and $v' \in V - X$. Since T' is a spanning tree, there is a unique path $u' \to u \to v \to v'$. Modify T' by adding edge e' and removing edge e. The resulting graph is still a spanning tree, and has lesser weight. q.e.d.

In general, G can have more than one minimum spanning trees. However, if the edge weights are distinct, then there is only one. The cycle property can be used to prove this easily.

We will analyse the Prim's algorithm for finding a minimum spanning tree of the given graph. We will also present the Kruskal's algorithm because it works well on fairly sparse graphs. However, the reader must be aware that a vast array of different algorithms is available for this problem.

6.1 Prim's algorithm

Prim's algorithm works by building the spanning tree one node and one edge at a time. Initially, the subgraph H is initialized to an arbitrary node s, and no edges. The algorithm terminates when all nodes in V have been added to the subgraph H. At each step, the algorithm chooses a minimum weight edge (u, v)

such that $u \in H$ and $v \notin H$. By this construction, it is easy to see that the subgraph H is a tree throughout the execution of the algorithm. Since all nodes in V are connected in H, H is indeed a spanning tree. And further because of the cut property, H is a minimum spanning tree.

The algorithm is almost always implemented with heaps. We will use a binary min-heap for our analysis. On each node w, two functions $N(w)$ and $\pi(w)$ are defined. $N(w)$ is initialized to ∞ and $\pi(w)$ to nil for all w at the start of the algorithm. π is stored in an array indexed by the node. And the heap in generated by inserting $< N(w), w >$ for all nodes w.

In general at every step of the algorithm, we add a new pair $< v, (u, v) >$ to H, except initially when we add only s. After a new node v and the corresponding edge (u, v) (except initially) have been inserted into H, he following update is applied: For every node z such that $(v, z) \in E$, run an update-key operation on the heap to set $N(z) = min(N(z), w_{(v,z)})$. If this results in an update, set $\pi(z) = v$. And then finally, perform the extract-min operation again.

The worst case analysis of this implementation is $O((m + n)logn) \sim O(mlogn)$ because each extract-min and update-key operation costs $O(logn)$; and there are exactly $n - 1$ extract-min operations, and at most $\sum_w d_w = 2m$ update-key operations, where d_w is the degree of node w. The time taken to setup the array π and the heap initially are both $O(n)$ and hence asymptotically smaller.

There are many algorithms for computing the minimum spanning tree that have better worst case performance than the Prim's algorithm. However, an average case analysis from Martel [6] of the Prim's algorithm reveals the reason behind its superior performance on dense graphs. The first step in this average case analysis is to notice that for any node u, the nodes that get added to H before u get added in the same order irrespective of the weights of edges incident on u. This is because for any node v such that $(u, v) \in E$, and v is added to H before u, N_v is updated only based on edges $(z, v) \in E$ such that $z \in H$.

So, if we randomly permute the edge weights of the edges incident on u, the expected number of update-key operations on u is $1 + 1/2 + 1/3 + ..1/d_u = H(d_u)\ \Theta(logd_u)$. This is because after the first update, the second update happens with probability $1/2$, the third update with probability $1/3$ and so on. Thus, the total expected number of update-key operations is $\sum_u \Theta(logd_u)$. Because $\sum_u d_u = 2m$, we can pessimistically maximise the expected number of update-key operations by using Lagrange multipliers to get a bound of $O(nlog(2m/n))$. Thus, the average case runtime of Prim's algorithm is $O(nlogn(1 + log(2m/n)))$.

As special cases of dense graphs, we see that if $m = nlogn$, the runtime is $O(mloglogn)$ and if $m = nlognloglogn$, it is $O(m)$. This clearly justifies the exceptional practical performance of Prim's algorithm on dense graphs.

6.2 Kruskal's algorithm

Kruskal's algorithm maintains a forest F, to which it adds an edge on every step, and finally produces a minimum spanning tree. The description of the algorithm is very simple: Sort the edges by their weights. And then keep adding the edges in ascending order of weight, unless a cycle is created.

The Kruskal's algorithm is implemented with the Union-Find data structure, which maintains the connected components of the forest F, and allows two operations (a) $Unite$: Unite two connected components, and (b) $Find$: Find the component that a node belongs to. Initially, each node belongs to its own component. Every time an edge $e = (u, v)$ is considered for addition to the forest, the algorithm firstly computes $c_u = Find(u)$ and $c_v = Find(v)$. If $c_u \neq c_v$, then the algorithm invokes $Unite(c_u, c_v)$, and adds e to F. It is easy to see by induction and the cycle property that the forest F at termination is a minimum spanning tree. Implementations of the Union-Find data strucure exist that can allow a find operation in $O(log^* n)$ amortized time, and unite in $\Theta(1)$ time. Because the algorithm requires $O(m)$ find operations, it runs in $O(mlog^* n)$ time.

7 Shortest paths in a graph

Consider a graph $G = (V, E)$, where every edge $e \in E$ has some non-negative weight w_e. G need not be connected. G could be directed or undirected. We are given two nodes s and t, and we need to find a

path $s \rightarrow u_1 \rightarrow u_2 ... u_k \rightarrow t$ of least weight. Evidently, this "shortest" path must be simple i.e. have no nodes repeated becase we can then discard the portion of the path between the repetition, and have a path that is not any "longer". The most intuitive algorithm for finding shortest paths in a graph is the Dijkstra's algorithm.

7.1 Dijkstra's algorithm

The Dijkstra's algorithm greedily computes the shortest path from s to a every node in a subset S of nodes until $t \in S$ or the entire connected component of s is explored, whichever happens earlier. The algorithm is implemented much like the Prim's algorithm for computing minimum spanning trees. On each node w, two functions $g(w)$ and $\pi(w)$ are defined. $g(w)$ is initialized to ∞ for all nodes except s for which it is set to 0. And $\pi(w)$ is set to nil for all w at the start of the algorithm. π is stored in an array indexed by the node. And the heap in generated by inserting $< g(w), w >$ for all nodes w except s. Here, $g(w)$ is intended to represent the length of the shortest path from s to node w, and $\pi(w)$ to represent the node preceding w in this shortest path.

In general at every step of the algorithm, we add a new pair $< v, (u, v) >$ to S, except initially when we add only s. After a new node v and the corresponding edge (u, v) (except initially) have been inserted into S, the following update is applied: For every node z such that $(v, z) \in E$, run an update-key operation on the heap to set $g(z) = min(g(z), g(v) + w_{(v,z)})$. If this results in an update, set $\pi(z) = v$. And then finally, perform the extract-min operation again. The correctness of this algorithm is easily proved by induction on $g(w)$. At the temination of the algorithm, $g(t)$ represents the length of the shortest path from s and The shortest path can be found out by backtracking from $\pi(t)$.

Since, the algorithm resembles the Prim's algorithm, except for the asymptotically cheaper $O(n)$ backtracking step, the worst case and average case runtimes are exactly the same [6].

8 Bipartite matching

Consider an unweighted bipartite graph $B = (U, V : E)$. A matching on B is defined as a set of vertex disjoint edges (or node pairs in other words). We wish to compute a matching in B, that has the maximum number of edges (i.e. of maximum size). There are two significant techniques that one can adopt in solving this problem: (a) Augmentation and (b) Push-relabeling. We will discuss results for the Hopcroft Karp algorithm that uses augmentation. This algorithm performs well for sparse bipartite graphs.

All augmentation based algorithms maintain a matching M that grows at each step, till it attains the maximum value. At any point during the execution of the algorithm, an unmatched node or edge is said to be free. An augmenting path is a path that starts at a free node, ends at a free node, and whose edges alternate between matched and free. Clearly, if B has an augmenting path P, then the symmetric difference $M \backslash P$ is itself a matching and has one more edge than M. So, by repeatedly finding an augmenting path in B, we can increase the size of the matching. The following theorem due to Berge guarantees that such a procedure should give us the maximum matching: If B has no augmenting paths wrt M, then M is a maximum matching.

Proof: Consider another matching M' of size greater than M. Then $M' \backslash M$ consists only of cycles and disjoint paths, on which the edges from M and M' alternate. This is true because each node in $M' \backslash M$ has degree at most 2, and M and M' are legitimate matchings.

The cycles must be of even length and consist of the same number of edges from M and M'. Since $|M'| > |M|$, there must be a path which has one more edge from M' than from M. Wrt M, this path must start from a free node, end on a free node and have alternating matched and free edges from M. Hence it is an augmenting path wrt M. But this is a contradiction because we assumed that B has no augmenting paths wrt M. q.e.d.

Consider the simple algorithm that grows a matching M by one, by finding *one arbitrary* augmenting path at each step. For a bipartite graph B, such an augmenting path must start from U and end in V.

Such a path can be found by DFS in time $O(m+n)$ where $m = |E|$. Since the size of the maximum matching is at most n, where $n = max(|U|, |V|)$, the running time is $O((m+n)n)$.

8.1 Hopcroft Karp algorithm

Hopcroft and Karp [7] improved the running time of the augment-one-at-a-time algorithm by repeatedly augmenting M with a maximal set S of shortest node-disjoint augmenting paths at each step. S is constructed by running a modified DFS on a *layered* graph L_M wrt M. We define L_M below and then prove that each augmentation step takes $O(m+n)$ time, and that the number of augmentation steps is $O(\sqrt{n})$ thereby proving that the Hopcroft-Karp algorithm runs in $O((m+n)\sqrt{n})$ time. Thereafter, we present the work of Motwani [8] which derives the average case runtime of this algorithm.

For any set X of nodes and Y of edges, $\Gamma_Y(X)$ is defined as the set of nodes adjacent to the nodes in X using only edges in Y. The graph L_M has an even number of layers starting 0. Call the set of nodes on the i^{th} layer as W_i and the set of nodes on all layers upto and including the i^{th} layer as Z_i. At layer 0, L_M contains all the free nodes in U. At any odd layer i, L_M consists of $\Gamma_E(W_{i-1}) - Z_i$. At any even layer i, L_M consists of $\Gamma_M(W_{i-1})$. It is easy to see that edges between any odd and the next (even) layer are matched; and the edges between any even and the next (odd) layer are free. Figure 1 illustrates the concept of a layered graph.

Given M, L_M can be constructed in $O(m+n)$ time. Notice that the nodes at even levels belong to U and those at odd levels to V. At each step in the Hopcroft-Harp algorithm, L_M is constructed to a level L no more than sufficient to contain a free node from V (which occurs at an odd level). Notice that in L_M any path from level 0 to L is an augmenting path. Thus, a maximal set of node-disjoint augmenting paths can be found by running DFS on L_M starting from level 0 nodes. Of course, to ensure that the paths are node-disjoint, additional mechanisms must be in place, but these are no more than constant overhead. Since DFS takes $O(m+n)$ time, each step of the Hopcroft-Karp algorithm runs in $O(m+n)$ time.

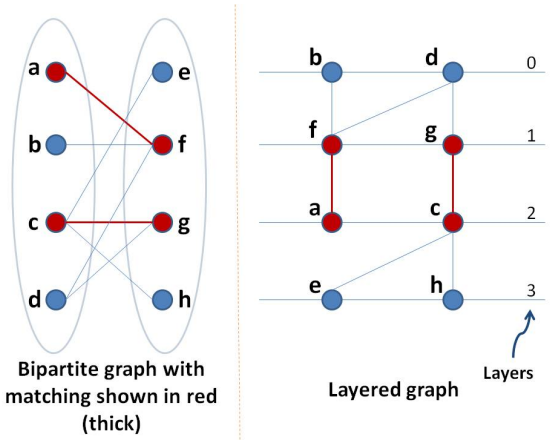

Figure 1: Layered graph for bipartite matching

Now, we can focus on characterizing the number of augmentation steps required by the algorithm. To do this, we prove that the length of the augmentation paths grows after each step. Then as a corollary, after $\sqrt{n}$ steps, the length of the augmentation paths must be at least $\sqrt{n}$, and there cannot be more than $\sqrt{n}$ such paths because these paths have to be node-disjoint. This would imply that the algorithm would run for at most $\sqrt{n}$ more steps. All in all, the algorithm runs for $O(\sqrt{n})$ steps.

Consider a matching M such that the shortest augmentation path wrt M has length k. And let S denote a set of maximal node-disjoint augmenting paths each of length k. We claim that after augmenting M by S, the length of the shortest augmenting path in the resulting matching M' greater than k.

Proof: Consider an augmenting path P wrt M'. If P is disjoint from S, then it has to have a length greater than k, otherwise S would not be maximal. So assume that P intersects with a subset F of t paths from S. Figure 2 illustrates this construction. In general, we will have $t+1$ paths as indicated by A, B, C and D, which are all augmenting paths wrt M. Hence, none of them can be shorter than k. From a simple geometric observation, one can infer that the sum of the lengths of these $t+1$ paths, which is at least $(t+1)k$ is less than the sum of lengths of all paths in F and that of P. Thus, we have $(t+1)k < tk + |P| \Rightarrow |P| > k$. q.e.d.

Now, we present the work of Motwani [8], which analyzes the expected number of steps in the above algorithm. We first consider the simplest model of random graphs and then present some definitions.

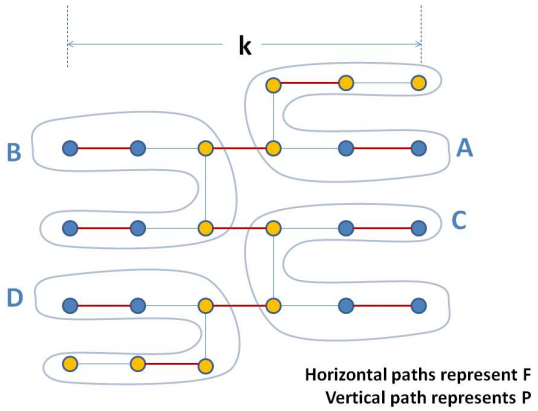

Figure 2: Augmentation wrt to M'

Denote $B_{n,p} = (U, V : E)$ a bipartite graph such that $|U| = |V| = n$ and each edge is picked with a probability p. Thus, the expected number of edges is n^2p and the expected degree Δ is np. Call the maximum degree Δ_m. The set U is referred to as the set of boys, and V the set of girls. We define $S(U) \subset U$, such that all nodes in $S(U)$ have a degree less than $\Delta_s < \Delta$. Let $L(U) = U - S(U)$. $S(U)$ is referred to as the set of small boys, and $L(U)$ the large boys. $S(V)$ and $L(V)$ are defined similarly. The following is the central claim of the runtime analysis: Let $B \in B_{n,p}$ such that $p > ln\, n/n$. Then the following properties hold with high probability for apprpriately chosen constants Δ_s, σ, λ and β.

1. $\Delta_m \leq \sigma\Delta$. i.e. the maximum degree is at most a constant greater than the average degree.
2. For $X \subseteq L(U)$ and and $\Gamma_B(X) > \lambda\Delta|X|$, $|X|$ need be no greater than n/Δ. i.e. for a lower-bounded adjacency set size, the size of the set of large boys is small.
3. For each $u \in U$, there is at most one $u' \in S(U)$ at a distance 2 from u. Moreover if there is a node $u' \in S(U)$ at a distance 2 from u, then there is exactly one path (length 2) from u to u'.
4. Two sets $X \subseteq U$ and $Y \subseteq V$ with more than n/β nodes each have at least one edge in between them.

Proof:

1. Let ψ_u denote the random variable equal to the number of subsets of size $\sigma\Delta$ in $\Gamma_B(u)$ for some $u \in U$. Then, because ψ_u is a non-negative integer variable, by Markov's inequality, $Pr(\psi_u > 0) \leq E(\psi_u) = \binom{n}{\sigma\Delta}p^{\sigma\Delta}$. Since ψ_u is positive iff $\Gamma_B(u) \geq \sigma\Delta$,

$$Pr(\Delta_m \geq \sigma\Delta) \leq n\binom{n}{\sigma\Delta}p^{\sigma\Delta} \leq (\tfrac{e}{\sigma})^{\sigma\Delta}$$

The rightmost inequality is true because $\binom{n}{k} \leq (\frac{ne}{k})^k$. Now, since $p > ln\, n/n$, we have $\Delta > ln\, n$, and therefore,

$$Pr(\Delta_m \geq \sigma\Delta) < n(\tfrac{e}{e^{ln\,\sigma}})^{\sigma ln\, n} = n^{-(\sigma ln\,\sigma - \sigma - 1)}$$

The RHS can be made arbitrarily small by choosing σ accordingly. q.e.d.

2. Consider a pair (X, Y) such that $X \subseteq L(U)$, $Y \subseteq V$, $|X| = k \leq n/\Delta, |Y| = \lambda\Delta k$ and $\Gamma_B(X) \subseteq Y$. Such a pair (X, Y) is called a bottleneck pair. Consider a bipartite graph B' in which each of the nodes in X has exactly Δ_s neighbors. Then, $Pr((X, Y)$ is a bottleneck pair in $B) \leq Pr((X, Y)$ is a bottleneck pair in $B')$.The RHS is given by

$$Pr((X, Y) \text{ is a bottleneck pair in } B) \leq \left(\frac{\binom{\lambda\Delta k}{\Delta_s}}{\binom{n}{\Delta_s}}\right)^k$$

Summing over all such pairs (X, Y), we get

$$Pr(\text{A bottleneck in } B) \leq \binom{n}{k}\binom{n}{\lambda\Delta k}\left(\frac{\binom{\lambda\Delta k}{\Delta_s}}{\binom{n}{\Delta_s}}\right)^k$$

By using the inequality $(\frac{n}{k})^k \leq \binom{n}{k} \leq (\frac{ne}{k})^k$, and large enough values of λ, we can upper bound this function by a super inverted-polynomial in n. q.e.d.

3. Let P_S denote the probability that a node $u \in U$ belongs to $S(U)$. There are two ways in which two nodes $x, y \in S(U)$ can be at a distance 2 from some $u \in U$. Either the two paths share a vertex in V or they don't. The probability of the first case is $n^4p^3P_S^2$ and that of the second case is $n^5p^4P_S^2$.

Now, P_S, the probability that there are at most Δ_s successes in n Bernouilli trials can be bounded by Chernoff bounds. And the desired probability can be driven down polynomially as desired. Similarly, the probability that there are two paths of length 2 can be bounded. q.e.d.

4. Let ψ be the random variable equal to the number of pairs of sets $A \subseteq U$ and $C \subseteq V$ such that $|A| = |C| = n/\beta$ and $\Gamma_B(A) \cap C = \phi$. Then,

$Pr(\psi > 0) \leq E(\psi) = \binom{n}{n/\beta}\binom{n}{n/\beta}(1-p)^{n^2/\beta^2}$

By the definition of e, for large n and because $\binom{n}{k} \leq (\frac{ne}{k})^k$, we get

$Pr(\psi > 0) < e^{\frac{n}{\beta}(2(1+ln\beta-\frac{\Delta}{\beta}))}$

The RHS can be made polynomially smaller to an arbitrary extent. Thus, the proof of all four properties is now complete.

Denote the set of all bipartite graphs that satisfy all the above properties as B_n^Δ. Then, for any $B \in B_n^\Delta$, and a non-maximum matching M, there is no augmenting path of length greater than $2L+1$, where $L = \Theta(ln\, n/ln\Delta)$. We prove this claim below but first notice that the truth of this claim would immediately imply that for any $B \in B_n^\Delta$, the Hopcroft Karp algorithm has at most $O(ln\, n/log\Delta)$ augmentation steps. Given that $Pr(B \notin B_n^\Delta) \leq n^{-1/2}$, in which case there are $\sqrt{n}$ augmentation steps, the expected number of augmentation steps for any bipartite graph is still $O(ln\, n/log\Delta)$. And in particular, if $p = 1/2$ i.e. the edges are chosen uniformly at random, then $\Delta = n/2$ and the expected number of augmentation steps is $O(1)$, thus resulting in $O(m)$ runtime overall for the algorithm. Now, we prove the claim about the augmentation path length.

Proof: Consider the layered graph L_M constructed by the Hopcroft Karp algorithm. We focus attention upto (at this point, arbitrary) level L. We also imagine a similar graph L'_M being constructed upto level L starting from free girls. If either L_M has a free girl or L'_M has a free boy, then we have an augmenting path of length at most L. So, we assume that this is not the case. If L_M and L'_M have a common node, then there is an augmenting path of length at most $2L$. So, we do not consider this case. We will prove that with the correct choice of L, there must be an edge between a boy in L_M and a girl in L'_M or vice versa, and therefore an augmenting path of length at most $2L+1$.

First, we focus on L_M. As before, call the set of nodes on the i^{th} layer as W_i and the set of nodes on all layers upto and including the i^{th} layer as Z_i. Define $W_i^L = W_{2i} \cap L(U)$, the set of "large" boys at layer $2i$; and define Z_i^L similarly. Call a girl matched to a boy in $S(U)$ as bad. Else call the girl good.

We know that there can be at most one bad girl adjacent to any $u \in L(U)$. Else there will be two small boys at a distance 2 from u. Thus, the set $\Gamma_B(Z_i^L)$ is adjacent to at most $|Z_i^L|$ bad girls. Also since Z_{2i} contains exactly $|Z_i^L|$ large boys, it contains at most $|Z_i^L|$ good girls. Thus, W_{2i+1}, the next level of girls contains at least $\Gamma_G(Z_i^L) - 2|Z_i^L|$ good girls. But this is also equal to $|W_{i+1}^L|$, the number of large boys in W_{2i+2}. This implies the following recurrence.

$$|Z_{i+1}^L| = |W_{i+1}^L| + |Z_i^L|$$
$$\geq \Gamma_G(Z_i^L) - |Z_i^L|$$
$$> (\lambda\Delta - 1)|Z_i^L|$$

The rightmost inequality holds because $B \in B_n^\Delta$ and also because Z_0^L, Z_1^L, Z_2^L cannot all be empty (otherwise some node in W_2 will have two small boys at a distance 2). This inequality implies that the number of large boys expands by a factor of $(\lambda\Delta - 1)$ at every level of boys. For some appropriate value of $L = \Theta(ln\, n/ln\Delta)$, we will have $|Z_{L/2-1}^L| > n/\Delta$. Indeed, if apply the same logarithmic argument to a set $\zeta \cup Z_{L/2-2}^L$ such that $\zeta \subset W_{L/2-1}^L$, then we can get that for some appropriate value of $L = \Theta(ln\, n/ln\Delta)$, $|Z_{L/2}^L| > \lambda n$.

We can apply the same reasoning for L'_M to arrive at a suitable value of L. Thereafter, we can use the fact that $B \in B_n^\Delta$ to prove that there must be a free edge between $Z_{L/2}^L$ in L_M and L'_M. And this leads to an augmenting path of length at most $2L+1$.

References

[1] Yehoshua Perl, Alon Itai, Haim Avni, *Interpolation Search - A lolglogN search* 1978.

[2] James Aspnes *Notes on randomized algorithms* 2011.

[3] Maria-Florina Balcan, *Notes on design and analysis of algorithms* 2011.

[4] Eva Tardos, Jon Kleinberg, Algorithm Design 2006.

[5] Donald Knuth, The art of computer programming, Volume 3, 2/e 2010.

[6] Chip Martel, *The expected complexity of Prim's minimum spanning tree algorithm* 2002.

[7] J.E. Hopcroft, R.M. Karp, *An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs, SIAM Journal on Computing 2 (4): 225231* 1973.

[8] Rajeev Motwani, *Average case analysis of algorithms for matchings and related problems* 1994.

[9] R.M. Karp, J.K. Lenstra, C.J.H. McDiarmid, A.H.G. Rinooy Kan, *Probabilistic analysis of combinatorial algorithms: An annotated bibliography* 1984.