Homework 9

Solution Key

Problem 1

Problem 2

Part 1 you divide each column by the sum of each column. You could use bsxfun, repmat, or a for loop.

Part 2 initializes a distribution p=[1;zeros(2999,1)], and then has a bunch of statements like p5=fr*p4. To get a good plot, something like plot(p5-p10) works, or basically just the difference between any pair that is far enough along on the random walk (though p9 and p10 may be so close that you get precision issues).

The image for part 3 should be three fuzzy blue clusters of points, each fairly well separated.

And for part 4, maybe some discussion about how random walks reveal interesting things about the structure of a graph that would be hard to see otherwise; and further, that eigenvectors reveal delicate information about random walks that just may be hard to figure out from random walks themselves (exactly the connection between random walks and eigenvectors is beyond the scope of this course); as long as your are at a point where you might be tempted to try the techniques here "in real life", then we're good.

Problem 3

The Matlab here involves just reordering pixels in an image. There are of course slick one-line ways to do this, but anything is fine. The 'econ' in the svd command tells Matlab to compute a small output, which saves your laptop time+memory. The output to part 2 should vaguely like the picture in the assignment, except the 16x16 image blocks will be ordered in descending order of importance, so the first row of image blocks will be the important ones. Again, the punchline here is about using sophisticated tools to reveal things you wouldn't know to look for otherwise; also, it gives us faith in the Fourier (or cosine) transform that we can recover something essentially the same via this purely statistical technique.

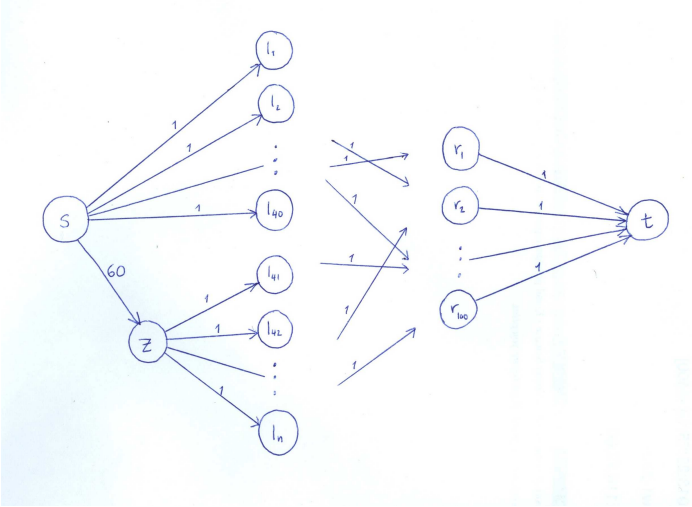
part 2:

We put this problem on the set because it is a powerful analytic technique with a broad range of applications. Most directly, this can be used not just for images but also for sound compression. Less directly, principal components analysis (PCA) can be used on many types of data, even discrete, and is commonly found in machine learning, pattern recognition, data mining to name a few. By preprocessing features with PCA before feeding them into some other algorithm (possibly a classifier like a support vector machine (SVM) or a statistical model like a hidden Markov model (HMM)) one can allow the more powerful algorithm to focus on the "important" parts of the data and thus do a better job than it would have on the unprocessed data.

The PCA basis looks similar to the cosine basis: the first elements of each are basically the same; the next two elements of the PCA look like a rearrangement of the two elements on the second diagonal of the cosine basis (possibly rotated); the next three elements of the PCA look like a

rearrangement of the 3 elements on the 3rd diagonal of the cosine basis; after this it becomes slightly harder, visually, to match up parts of the two bases. But both go from representing coarse detail to representing fine detail. Both are *orthonormal*, meaning that each basis vector (16×16 image chunk) from one of the two bases has a dot product of 0 with every other basis vector in its basis.

Problem 4



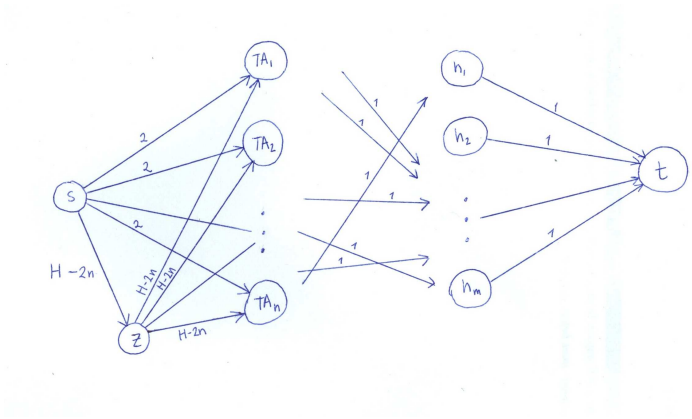
Part 1

Consider the above graph with capacities on the edges. Formally, given a marriage problem, construct the corresponding graph as follows: create a node for each person, with an edge of capacity 1 connecting each marriageable couple; create a sink node t that has an edge of capacity 1 from each person on the right; create a source node s that is connected with edges of capacity 1 to each of the *first* 40 people on the left; create a node z that has capacity 60 from s, and is connected to each of the remaining people on the left with capacity 1.

We first note that any valid marriage plan corresponds to an $s \to t$ flow of 100: send one unit of

flow across each edge corresponding to each of the 100 marriages; send one flow from each of the married people on the right to the sink; each of the first 40 people on the left, who are married by assumption, receives 1 flow from the source; each of whichever 60 remaining people on the left are married receive 1 flow from z, which receives 60 flow from the source. Further, because the cut through s has capacity 100, the maximum flow in the graph is hence exactly 100.

We thus propose the following algorithm: run a max-flow algorithm on the constructed graph, and marry those people with flow connecting them. As just shown, this max-flow equals 100; further, the algorithms from class produce integer flows given integer capacities. Thus our algorithm outputs 100 marriages. Since each person on the left has capacity 1 going into them, and each person on the right has capacity 1 going out of them, each person may be involved in at most 1 marriage. Since the total capacity to all the people on the right except the top 40 is 60, there can be at most 60 marriages from this group; however, since there are 100 total marriages, and only 40 people in the top group, we have just shown that our algorithm outputs 100 total marriages, of which exactly 40 are from the top group on the left.



Part 2

(This is the solution to the extra credit version of the problem)

We note that you can solve this problem strictly from the results of the previous part. For each TA, make a large number of *clones* of her, and for two of her clones, mark them as *needing* to have office hours. Then express this problem as an instance of the marriage problem from the previous part: each clone is on the left, each hour is on the right, and a clone can "marry" an hour if that person is free to hold office hours then; each of a person's two clones who needs an hour is marked as needing a marriage, and is placed in the top left. Thus we have adapted the results of the previous part, with the total hours H replacing 100, and the number of mandatory marriages, $2n$, replacing 40.

In the diagram above, we have merely merged all of a person's clones back into one person again.

Problem 5

1. Using one multiprocessor with one thread means each 4-byte memory request from the thread results in a separate memory transfer. Thus $128/4 = 32$ memory transfers are required to read 128 bytes, using $32 * 200 = 6400$ns total time. If instead we use one warp of threads, all 32 threads can access different contiguous elements in parallel, necessitating only a single access for a total of 200ns.

2. Both of these resources are split evenly amongst the $32 = 2^5$ threads of each of the 32 warps. So each thread gets $2^{16}/2^{10} = 2^6 = 64$ registers and $32K/2^{10} = 2^{15}/2^{10} = 32$ bytes of shared memory.

3. (a) If all elements are in the same bank they must be accessed in serial in 32 access cycles at 1ns for a total of 32ns.

   (b) If all elements are in different banks they can be accessed in parallel in a single access cycle for a total of 1ns.

4. See 6.

5. See 6.

6. Execute with 128×128 blocks of 1024 threads each:

```
transpose(Input, Output)
  xblock <- blockID mod 128
  yblock <- blockID / 128
  locI <- xblock*32 + yblock * 4096 * 32
  locO <- yblock*32 + xblock * 4096 * 32
  x <- threadID mod 32
  y <- threadID / 32
  locI <- locI + x + 4096*y
  locO <- locO + x + 4096*y

  shared array: S[1056=32*33]
  S[x+33*y] <- Input[locI]
  syncThreads()
  Output[locO] <- S[y+33*x]
```

7. The first 4 lines assign 32x32 sub-matrices and global memory base pointers to each block of 32 warps and is only necessary for part 6. The next 2 lines assign unique x,y coordinates to threads within a block and is necessary for parts 4, 5, and 6. The next 2 lines assign global memory I/O coordinates to each thread. The stride would be 32 instead of 4096 for parts 4 and 5. The last 4 lines do the work. It would be more natural to use 32 instead of 33 but using 33 offsets the memory bank assignment so that reading/writing in both column and row major order does not cause memory bank conflicts.

   Our code performs a transpose by reading the sub-matrices into shared memory in the correct order and writing them back out to global memory in transposed order.

8. We *expect* our algorithm to use all 200GB/s of the card's memory bandwidth, because we were very careful to access memory in the optimal way, and there is almost no computation needed to transpose a matrix, so if *anything* could achieve 200GB/s, our algorithm should.

   To justify this, we try to estimate how much memory bandwidth our algorithm would use if we ignore the 200GB/s limit.

   Each thread reads and writes one 4-byte element from global memory, and thus each warp reads and writes one 128 byte chunk. When a warp first requests 128 bytes from global memory, it will wait 200ns for the request to be answered, and meanwhile other warps will swap in and continue their execution. Other than global memory access, nothing in our code should take more than a few nanoseconds. The important thing is that, even though there are only 128 cores per multiprocessor and hence we can be doing arithmetic on behalf of only 4 warps at any instant, *all* the warps can have memory requests pending at the same time. Also, while a thread cannot proceed beyond a global memory read until it receives the value from global memory, a thread does *not* have to wait for its global memory *writes* to "complete", because there is no data to wait for. Thus, except for the small amount of time doing address arithmetic, we would expect each thread block to be finished processing in essentially 200ns. Since there are 8 multiprocessors processing thread blocks at once, this leads to a total time of $\frac{128\times128}{8}200ns = 0.0004096s$, which implies a memory transfer rate of $\frac{4096\times4096\cdot4B\cdot2}{0.0004096s} \approx 305GB/s$. Since this exceeds 200GB/s, we conclude that the memory bus is the limiting factor, 200GB/s is the total throughput, and our algorithm takes $\frac{4096\times4096\cdot4B\cdot2}{200GB/s} = 0.000625s$.