Corrections From last class:

- Dijkstra’s algorithm does not handle negative edge weights Although it may still give us the right answer sometimes...

- Psudo code:

```
Dijkstra( graph G , vertex v0)
 {
 // initialization

        Priority Queue q  = new PriorityQueue();
        D[v] = 0;
        q.push(v0,D[v0]);
        for all vi != v0
        {
                D[vi] = infinity;
                q.push(vi,D[vI]) // Should have been vi not v0 here..
        }

 //start of algorithm

        vertex temp;
        while(!q.isEmpty())
        {
                temp =  q.getMin();
                for all vertices z adjacent to temp
                {
                        if(D[temp]+weight(temp,z) <D[z])
                        D[z] =  D[temp]+weight(temp,z);
                        q.changeKey(z,D[z]);
                }
        }
        return D[ ];
 }
```

Shortest Paths:(continued)

- The Bellman-Ford Algorithm:

  i) When it is it applicable?

  ii) Does it handle negative cycles?

  iii) Correctness and Time Analysis

- The A* Algorithm

  i) How does it differ from the other two?

  ii) Special properties

  iii) Time analysis

When is it Applicable?

The Bellman-ford algorithm is another shortest path algorithm that finds the shortest path from one vertex to all othere vertices. However, in this case we can have negative weight edges, the draw back is that the edges must now be directged edges since an undirected edge constitutes a negative cycle (as traversal can be done back and forth between the two vetices that are joined by it!)

Does it handle negative cycles?

Well the answer is yes and no, while it will not find the correct Answer if there exist any negative cycles, it will detect the existace of one !

Anything else?

while the Bellman-Ford algorithm does use edge relaxation it does not use it with the greedy method as the greedy method does not apply in this context. This is because instead of choosing the minimum local solution at each node, we simply preform edge relaxation on each for each edge and we are either left with the optimal solution after n-1 edge relaxations or there is a negative Cycle.

The Algorithm:

```
boolean Bellman-Ford(graph G, Vertex v0)
{
    // Initialization
    boolean hasNegativeCycle = false;
    D[v0] = 0;
    For all vi != v0
    {
        D[vi] = infinty;
    }
    // The magic starts here
    For i = 1 to n-1
    {
        For every edge (vi,z) out of vi
        {
            if ( D[vi] + Weight(vi,z) < D[z] )
                D[z] = D[vi] + Weight(vi,z)
        }
    }
    mark solution;
    // nth iteration;
    for every edge (vi,z) out of vi
    {
        if ( D[vi] + Weight(vi,z) < D[z] )
            D[z] = D[vi] + Weight(vi,z)
    }
    if solution is same as before
        return hasNegativeCycle;
    else
        return !hasNegaiveCycle;
}
```

Correctness:

let di(v,u) denote the distance from v to u using at most i edges after the ith iteration we have that either
di(v,u) = di-1(v,u) or di(v,u) = di-1(v,z) + weight(z,u) for some vertex z. So, after the ith iteration we have
D[u] = di-1(v,u) = di(v,u)
or
D[u] = D[z] +weight(z,u) = di-1(v,z) + weight(z,u)
as we can see if D[u] = di-1(v,u) before the ith Iteration then D[u] = di(v,u) after the ith iteration.
(in otherwords D[u] only goes down).
Obsurve that in a shortest path from some v to u we have at most n-1 edges..
So,
if a edge can be edge relaxed after n-1 iterations then there must be some vertex repeated.
but since the distance of an edge to it self using zero edges is zero, then we must have a negative cycle!
else, we have an optimal solution..

Time Analysis:

As we can clearly see the number of edge relaxations preformed is n*m where m is the number of edges.
recall for a directed graph Sum(deg(vi), for all vi) = number of edges, in worst case the number of edges is n *(n-1) notice that it is twice the number of an undirected graph. So , we have that this case the algorithm is O(n^3). In a best case senarion we have that the number of edges is n-1 in which case the algorithm is O(n^2).

last comment:

while the Bellman-Ford Algorithm is slower than dijkstra'a algorithm it can handle negative edges and detect negative edge cycles this added functionality makes the Bellman-Ford algorithm usefull in networking problems where you must have negative edges..

The A* Algorithum:

- What is it?

  A* algorithm is a search algorithm that uses heuristic Knowledge as well as actualll path costs to find a minimal path to its goal if it exists. And does so efficiently if the graphs heuristic function generates values that are monotonic..

- Mono who's it now??

  Monotonicity means that for every connected node in the graph the difference in the generated heuristic values do not over estimate the actual value to the goal node.

- Ok so what if it doesn't satisfy the mono thinga magig?

  Well if the heuristic values don't satisfy the monotonicity condition then we must maintain 2 diifferent priority queues(usually called open and closed, as opposed to, just one if it had satisfied it.

- Sounds good but where would you use it? isn't Dijksta's better?

  The A* algorithm is used in many applications like tile based games its also is used in AI.
  this algorithm differes from dijkstra's algorithm because unlike dijkstra's the algorithm knows where its going and take advantage of previous knowledge in the form of the heuristic values to get there.

Show me the money:

```
A* algorithm(Graph g,v0,vf)
{
    // input a graph g with edge weights and estimates, start point,end point
    PriorityQueue open = new PriorityQueue();
    PriorityQueue Closed = new PriorityQueue();
    open.push(v0,v0.estimate+0);
    vertex min;
    while (!open.isEmpty())
    {
        min = open.getMin()
        if (min == vf)
            return min.currentHeuristic;
        else
        {

            for all vi addjacent to min
            {
                calculate new heuristic (vi);  // a form of edge relaxation.
                if (open.has(vi)&& vi.currentHeuristic < vi.newHeuristic)
                {
                    open.remove(vi);
                    closed.push(vi);
                }
                else if (closed.has(vi)&& vi.currentHeuristic > vi.newHeuristic)
                {
                    closed.remove(vi);
                    open.push(vi);
                }
                else
                    open.push(vi, vi.heuristic);
            }

            opent.remove(min);
            closed.push(min);
        }
    }
}
```

Examples Please!!:

http://www.cs.ubc.ca/labs/lci/CIspace/

Correctness:

We can see Intuatively that A* consideders the fewest nodes and is correct of any search algorithm if we recognize that A* uses an underestimate for the actual distance and tries to achive. When the search ends then the actual cost is less than the path through any node on the open queue. So, A* will never overlook a lower cost path to the goal => it is correct.
lets assume that a search algorithm B uses the same heuristic knowledge of A* but with less nodes. We See B will terminate with a path that has a cost more that the estimated cost on the open queue since it is only conserned with number of nodes thus in this context it is not correct. thus A* considers the least number of nodes required.

Time Complexity:

Time Complexity of A* is log(n) if we repeat the algorithm from v0 to all vi we have O(n log (n)) time to find the shortes paths..

Refences:

- Amits game programming information http://www-cs-students.stanford.edu/~amitp/gameprog.html#Paths
- Algorithm Design Foundation, Analysis, and Internet Examples. Michle T. Goodrich, Roberto tamassia. chapter 7
- CI space Tools for Learning and Artifical Intelegence. http://www.cs.ubc.ca/labs/lci/CIspace/
- Igor's notes form last year..