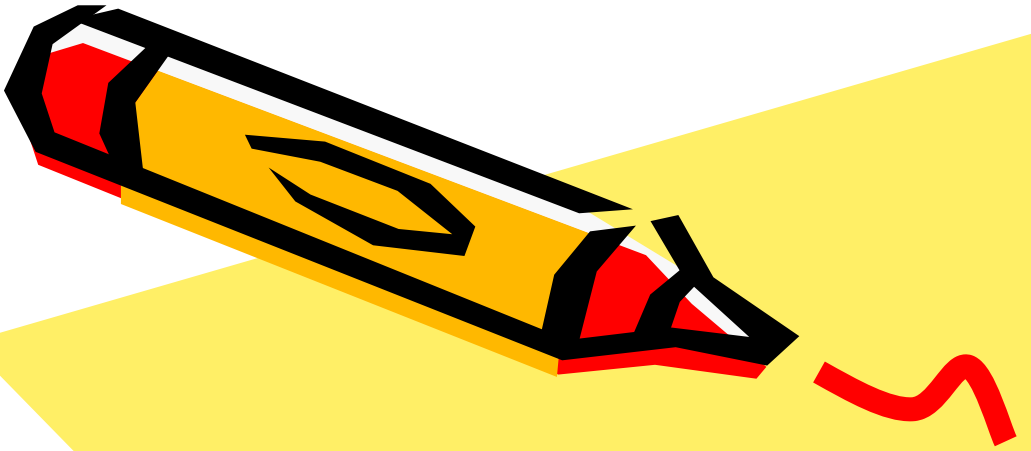
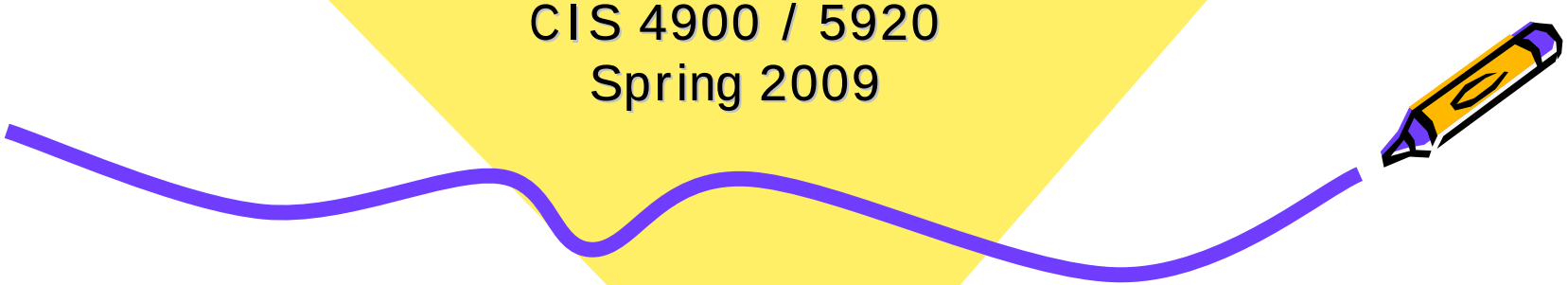


Dijkstra's Algorithm for Single-Source Shortest Path Problem

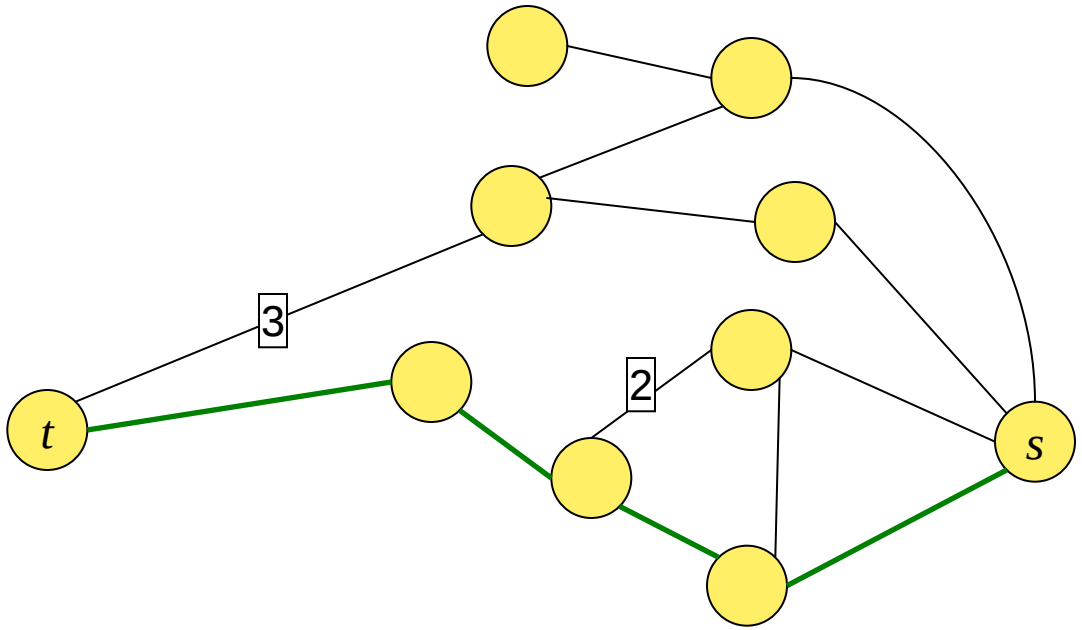
Programming Puzzles and Competitions
CIS 4900 / 5920
Spring 2009



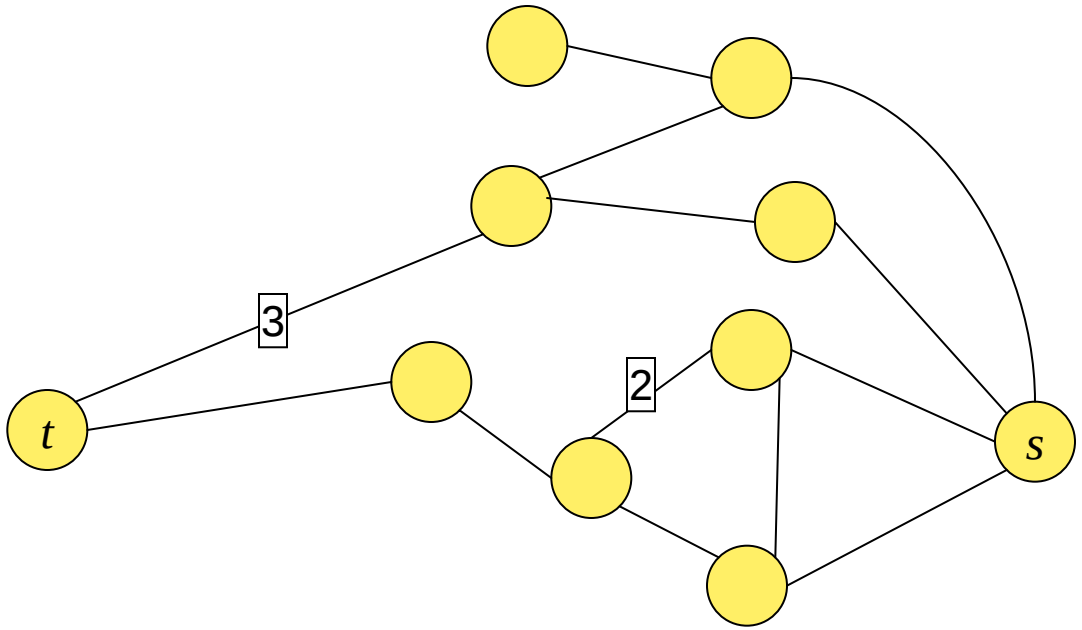
Outline

- Dijkstra's algorithm
- How to code it in Java
- An application to a problem on the FSU ACM spring 2009 programming contest

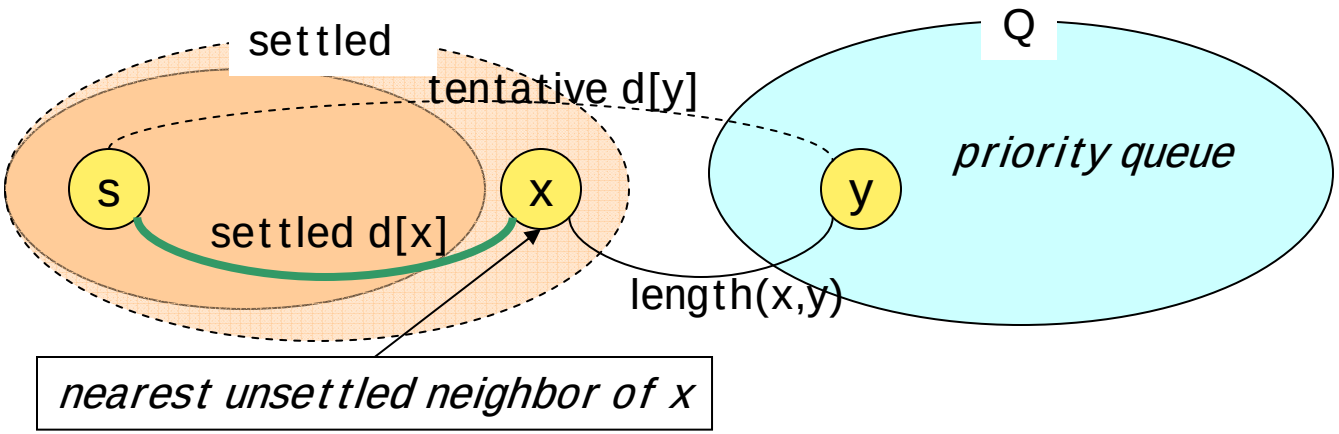
Point-to-point Shortest Path Problem



Point-to-point Shortest Path Problem

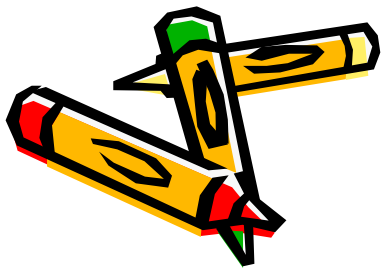


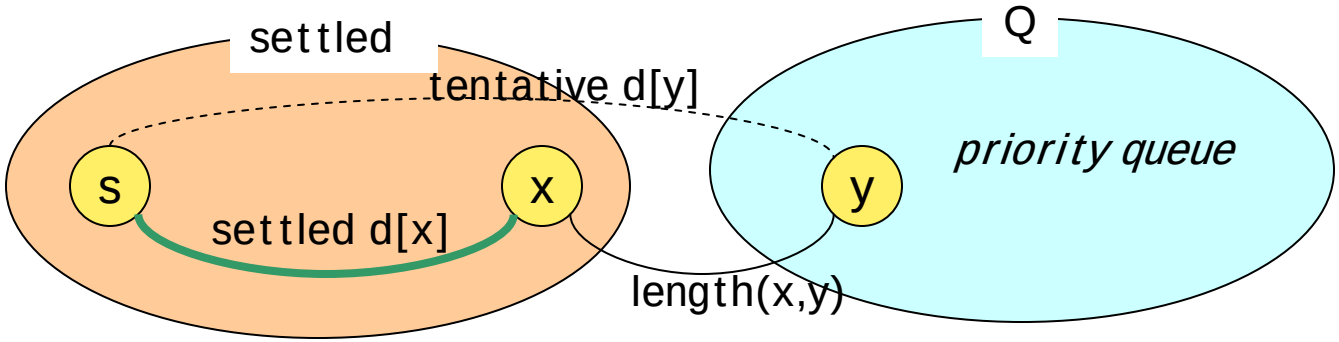
Dijkstra's Idea



1. Shortest distance from s to all nodes initially "unsettled".
2. Shortest distance to s is zero. Tentative distance to others is ∞.
3. Put all nodes in queue ordered by tentative distance from s.
4. Take out nearest unsettled node, x. Settle its distance from s.
5. For each unsettled immediate neighbor y of x
6. If going from s to y through x is shorter than shortest path through settled nodes, update tentative distance to y.
7. Repeat from step 4, until distance to destination is settled.

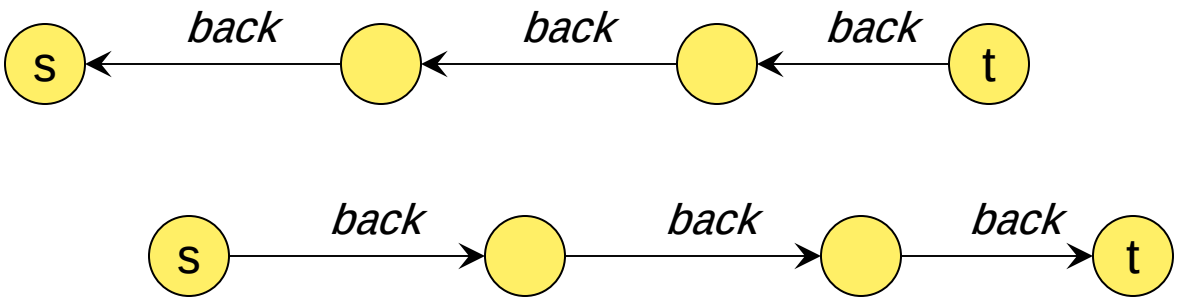
```
∀x ϵ V: d(x) = ∞; settled = Ø;                          } Θ(V)
Q = V; d(start) = 0;                                    } Θ(VlogV)
while (Q ≠ Ø) {
   choose x ϵ Q to minimize d(x);                       } Θ(logV)
   Q = Q – {x};                                         } Θ(logV)
   if (x==dest) break;
   settled = settled ∪ {x}; // d[x] is shortest distance to x
   for each unsettled neighbor y of x {                 } Θ(ElogV)
      if (d(y)>d(x) + len(x,y)) {
            d(y) = d(x])+ len(x,y);
            back(y) = x;}}}
```





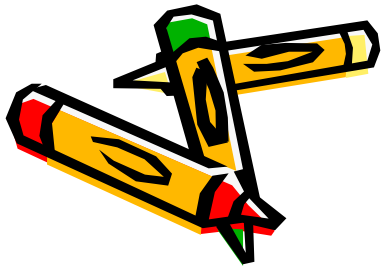
To extract path

1. trace back-links from destination to source, reversing them as we go
2. traverse reversed links from source to destination, to obtain a shortest path



To get minimal spanning tree

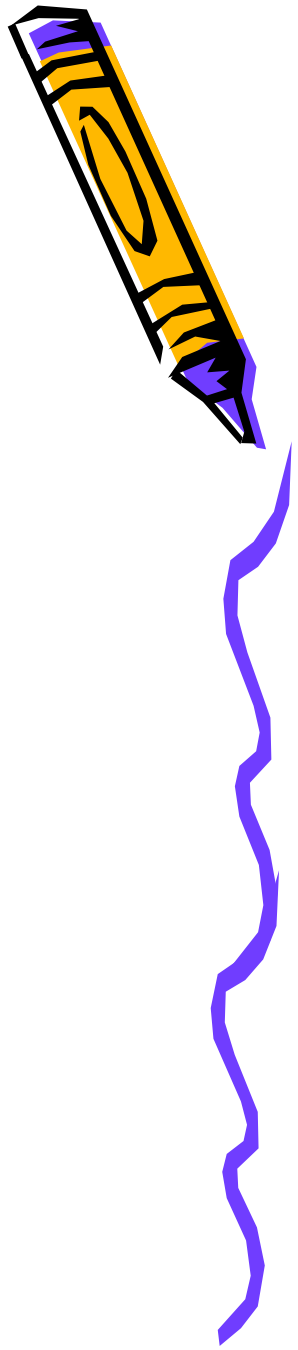
- Run until all nodes are settled
- Reverse all links



Example Application

- Problem from spring 2009 FSU local ACM programming contest
  http://www.cs.fsu.edu/~baker/pc/city/fsu0409contest.pdf
- Imaginary "city" is grid of squares
- Special rules about direction of travel between squares
- Find shortest path between two specified points

Movement Rules

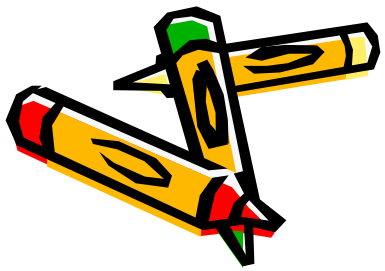


Example for n=4 :

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

From block x:

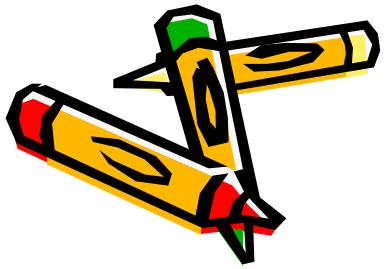
- x mod n = 0 → may move N or S
- x mod n = 1 → may move NE or SW
- x mod n = 2 → may move E or W
- x mod n = 3 → may move NW or SE

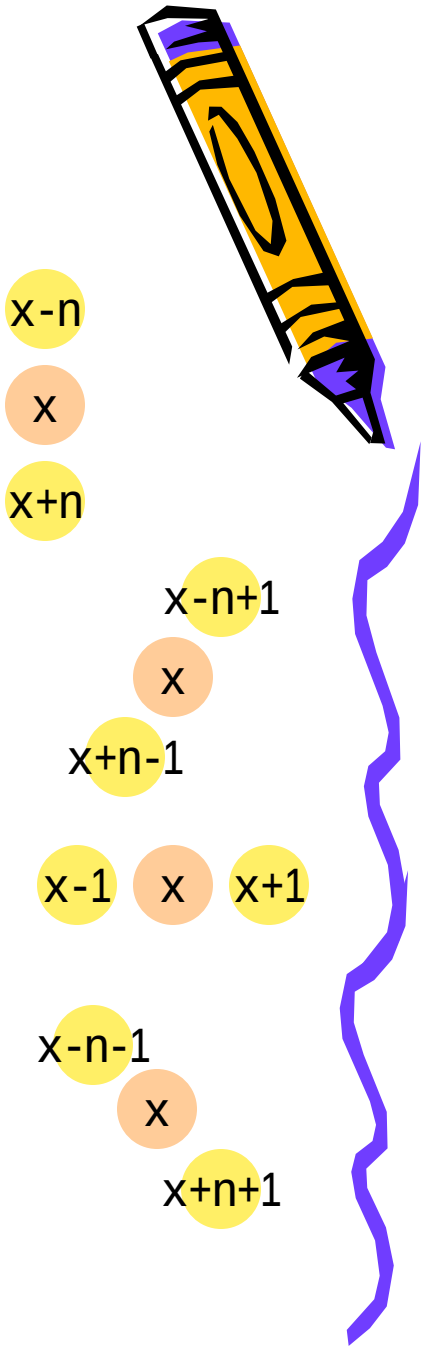


Movement rules

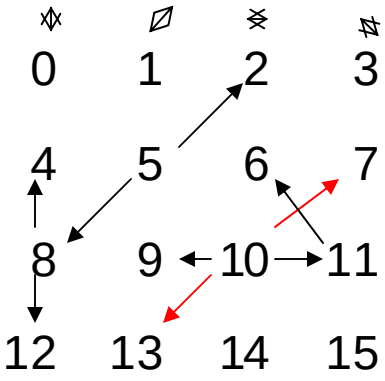
x mod n = 0 → may move N or S

- x mod n = 1 → may move NE or SW
- x mod n = 2 → may move E or W
- x mod n = 3 → may move NW or SE





Read problem again.



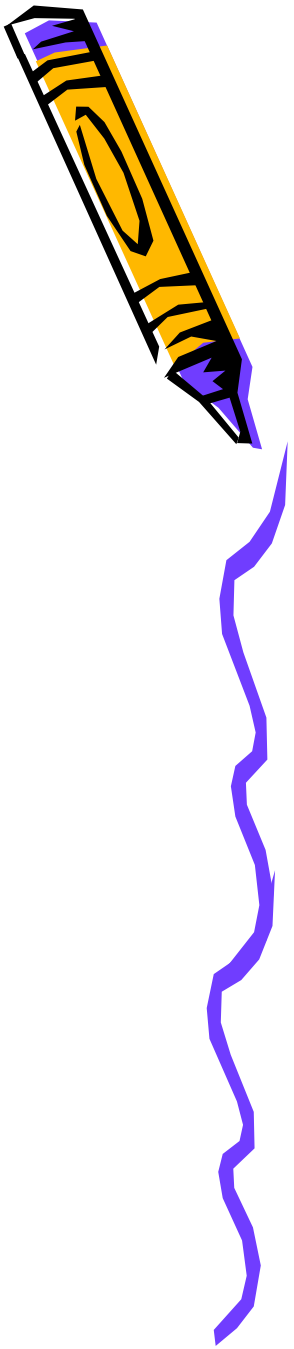
This example is inconsistent with the rule.

Assume the error is in the example?

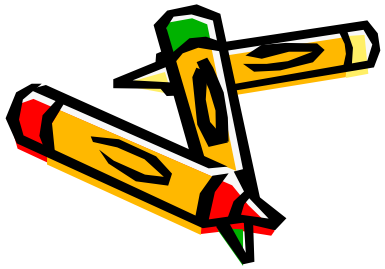
Ask the judge.

``For example, suppose n=4. If you are currently in block 8, you may move to block 4 and 12.
If you are in block 5, you may move to block 2 and 8.
If you are in block 10, you may move to block 7 and 13.
If you are in block 11, you may move to block 6.
Note that you may move to only one neighboring block if the other block does not exist."

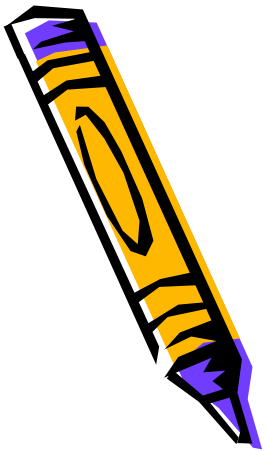
Designing a Java implementation



- How to represent nodes?
  - class? – too cumbersome for time limit
  - so, use integers 0 .. V-1, for V = n * n
- How to represent edges?
- How to represent distance?
- How to implement Q?



Edge representation



- Adjacency list is most efficient
- Avoids looking at non-edges
- Reduces from $O(V^2 \log V)$ to $O(E \log V)$
  - How to implement an adjacency list?

Simple special case

- In this case, number of edges per node seems limited to 2

  int neighbor[][] = new int[V][2];

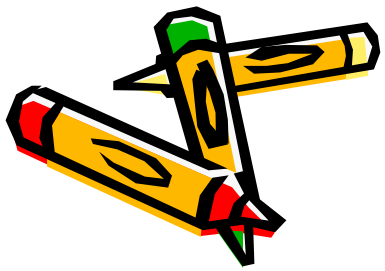
  neighbor[x][0] = first neighbor

  neighbor[x][1] = second neighbor

- What if less than two edges?

  neighbor[x][i] = -1

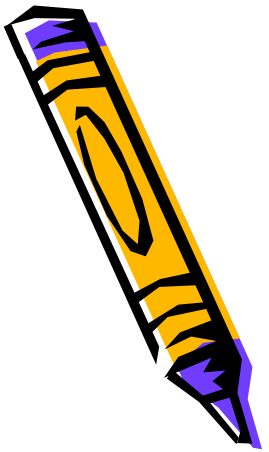
  - but now we need to check for this case



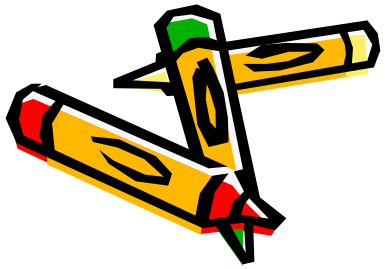
Setting up neighbor array.

```
for (int x = 0; x < V; x++) {
 switch (x % n) {
 case 0:
    if (x-n >= 0) neighbor[x][0] = x-n; // N
    if (x+n < V) neighbor[x][1] = x+n;   // S
    break;
 case 1:
    if ((x-n >= 0) && (x % n < n-1))
         neighbor[x][0] = x-n+1; // NE
    if ((x+n < N) && (x % n > 0))
         neighbor[x][1] = x+n-1; // SW
 ...etc.
}}
```

Alternatives



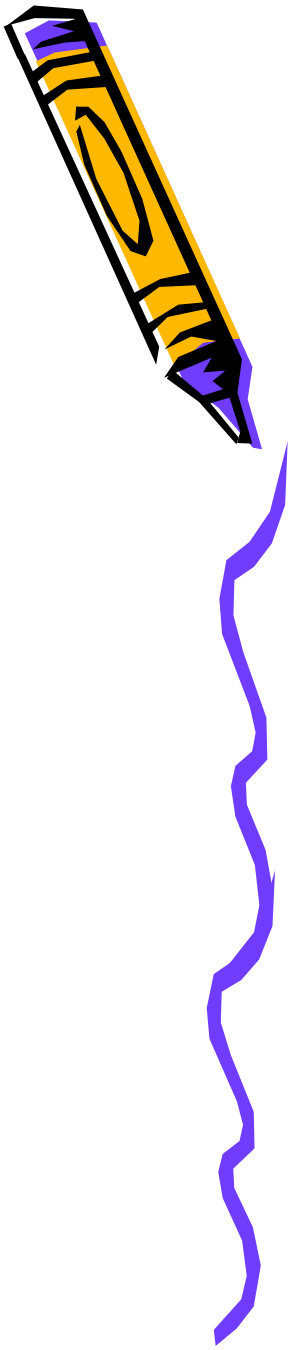
- array of arrays
  - saves -1 check, but need code to create sub-array of correct length
- implicit representation, using a function (or iterator)
  - e.g. int neighbor(x,i){ ...}
  - maybe a good idea, but estimate of coding time seems greater



How to represent settled?

```
boolean settled[] = new boolean[V];
for (i = 0; i < V; i++) settled[i] = false;
```

How to represent distances?

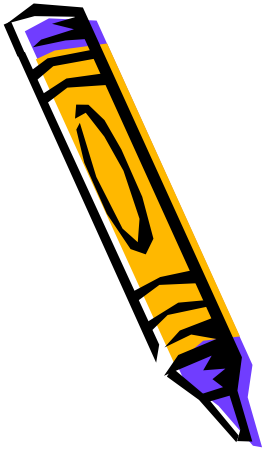


- int d[] = int[V];
- How to represent ∞?

```
for (i=0; i < V; i++)
    d[i] = Integer.MAX_VALUE
```

  - watch out for overflow later!

How to represent Q?

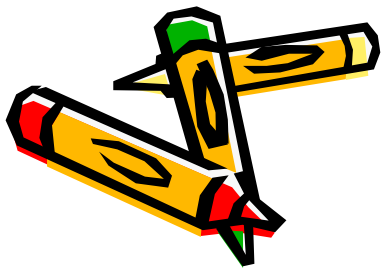


A. Roll your own priority queue?
B. Use Java utility library?
   - takes less time to code
   - no debugging time
   - if you know how to use it!

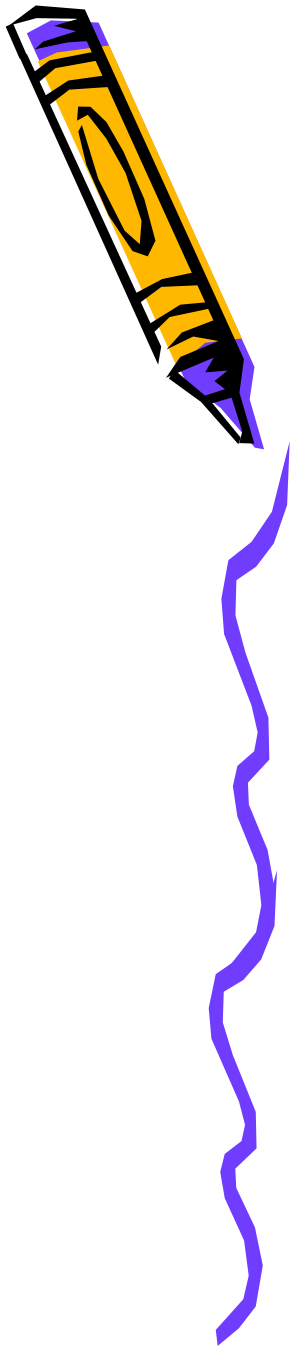
http://java.sun.com/javase/6/docs/api/java/util/PriorityQueue.html

Setting up priority queue.

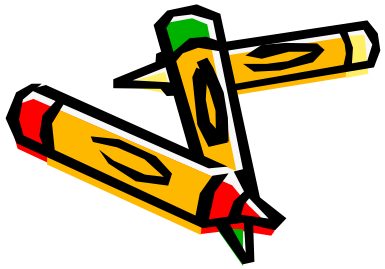
```
Comparator<Integer> shortestDistance =
    new Comparator<Integer>() {
    public int compare(Integer L, Integer R) {
      if (d[L] > d[R]) return 1;
      if (d[L] < d[R]) return -1;
      if (L > R) return 1;
      if (L < R) return -1;
      return 0; } };
PriorityQueue<Integer> q =
    new PriorityQueue<Integer>(N,
                               shortestDistance);
```

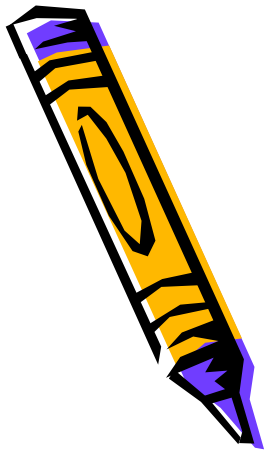


A literal coding of abstract algorithm



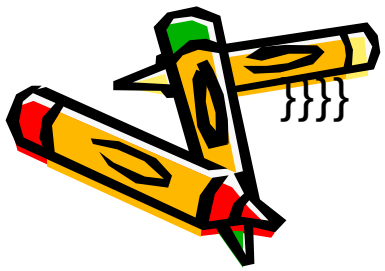
```
// ∀x ϵ V: d(x) = ∞; settled = Ø;
for (i = 0; i < V; i++) {
    d[i] = Integer.MAX_VALUE;
    settled[i] = false;
}
// Q = V; d(start) = 0;
for (i = 0; i < V; i++) q.add(i);
d[start] = 0;
```





```
// while (Q ≠ Ø) {
while (! q.isEmpty) {
   // choose x ϵ Q to minimize d(x);
   Q = Q – {x};
   x = q.poll();
   if (x==dest) break;
   // settled = settled U {x};
   settled[x] = true;
   // for each unsettled neighbor y of x {
   for (int i = 0; i < 2; i++) {
        y = neighbor[x][i];
        if ((i != -1) && ! settled[y]) {
            // if (d(y)>d(x) + len(x,y)) {
            if (d[y]>d[x] + 1){
                // d(y) = d(x)+ len(x,y);
                d[y] = d[x]+1;
                // back(y) = x;
                back[y] = x;
```

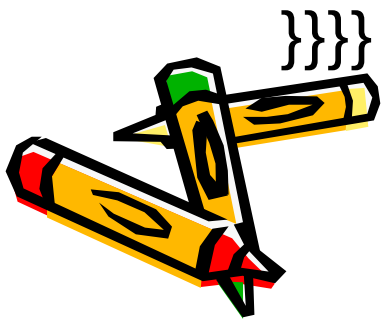
What's wrong with this?



Q details

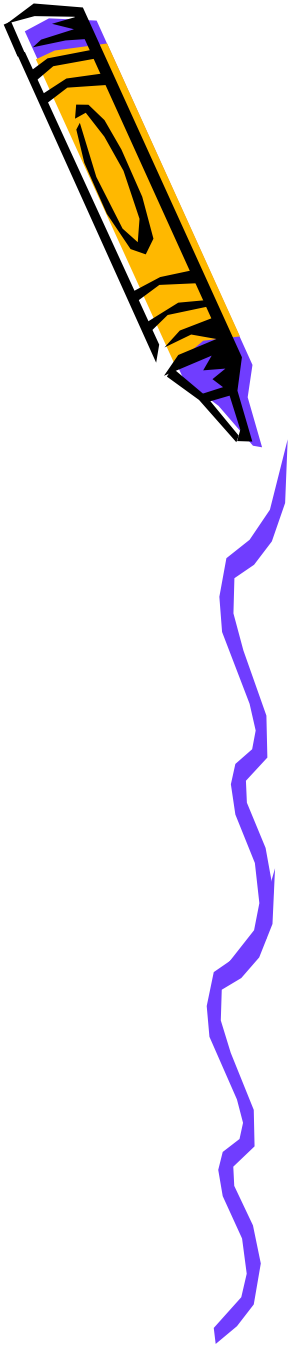
- Need to re-insert nodes in priority queue when priorities change
- Does re-insertion require deletion first?
  - Java documentation does not seem very clear on this, but
  - an experiment shows that repeated insertion will create duplicates.

```
while (! q.isEmpty) {
   x = q.poll();
   if (x==dest) break;
      settled[x] = true;
      for (int i = 0; i < 2; i++) {
         y = neighbor[x][i];
         if ((i != -1) && ! settled[y]) {
            if (d[y]>d[x] + 1){
               d[y] = d[x]+1;
               back[y] = x;
               q.remove(y);
               q.add(y);
}}}}
```

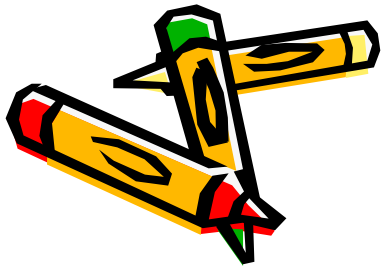


Remove and re-insert nodes with changed distance.

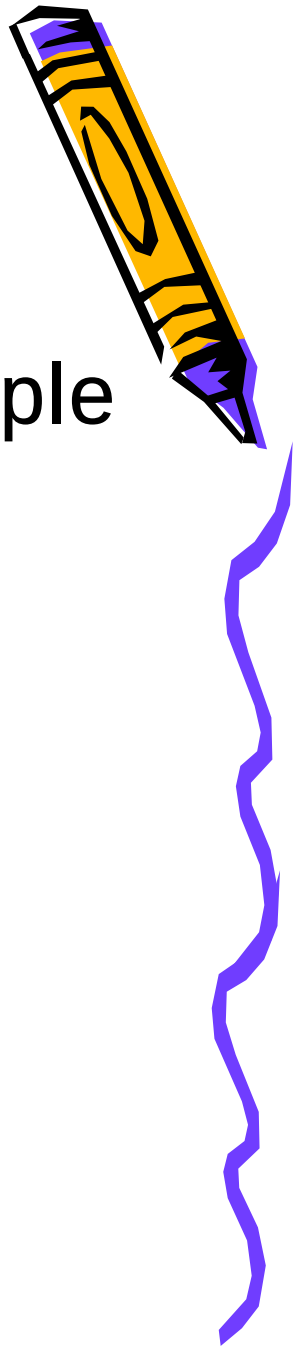
Simplify initialization, avoid visiting disconnected nodes.



```
for (i = 0; i < V; i++) {
    d[i] = Integer.MAX_VALUE;
    settled[i] = false;
}
// for (i = 0; i < V; i++) q.add(i);
q.add(start);
d[start] = 0;
```



We run program. It fails.



- Fails to find any path on given sample input:

  16 99 5

- Look at sample output:

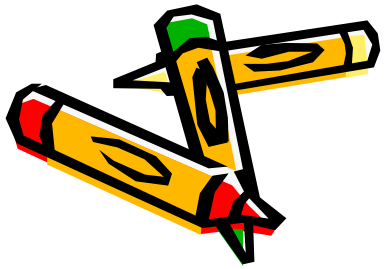
  99 116 100 84 68 52 36 20 5

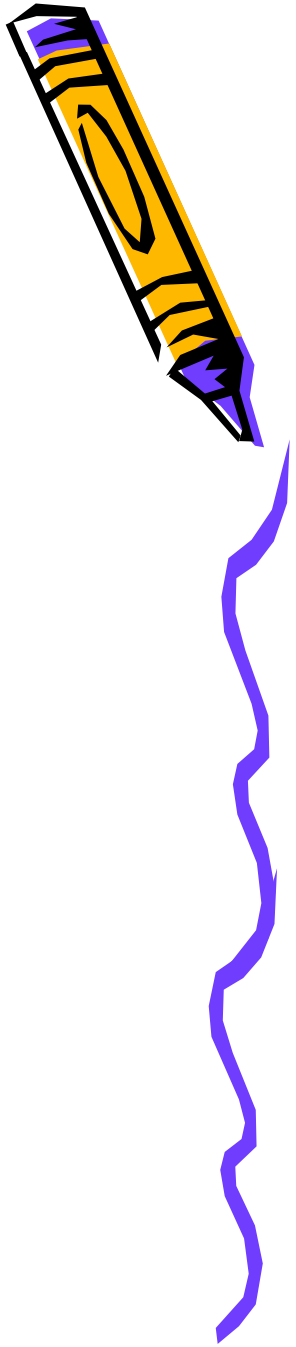
Study example in detail

Modulus seems to be 4 rather than N.

20 mod 4 = 0 so can only move to N or S, so intent seems to be that edges are bidirectional

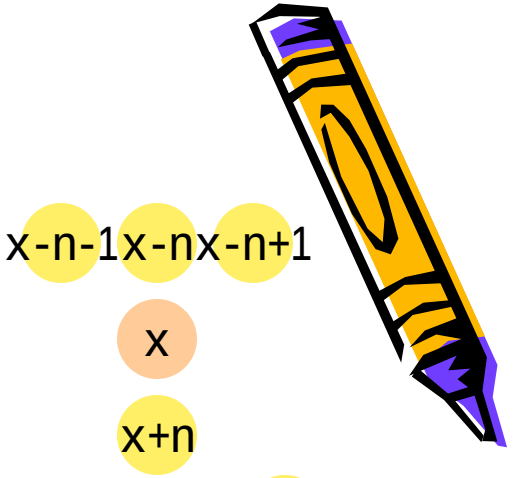
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255



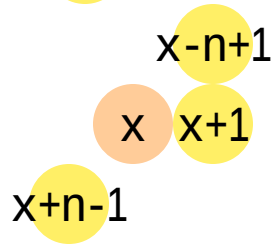


Movement rules

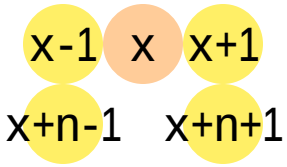
x mod n = 0 → may move N or S or NW or NE



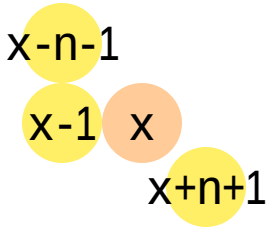
- x mod n = 1 → may move NE or SW or E



- x mod n = 2 → may move E or W or SW or SE

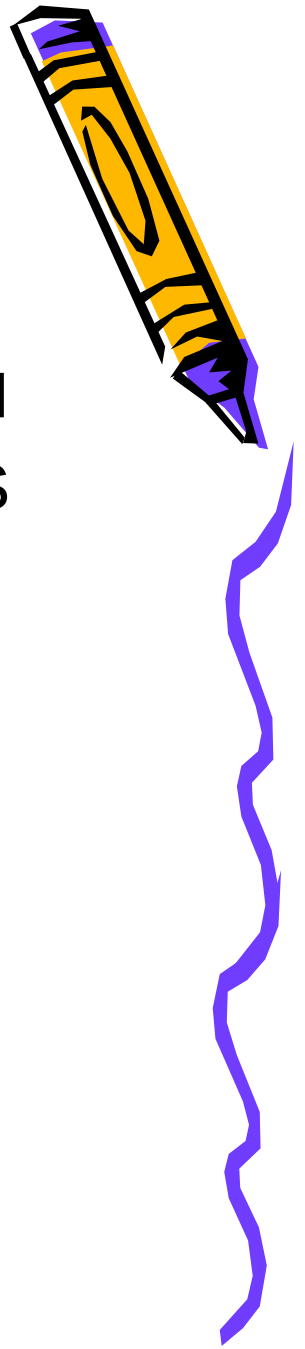


- x mod n = 3 → may move NW or SE or W

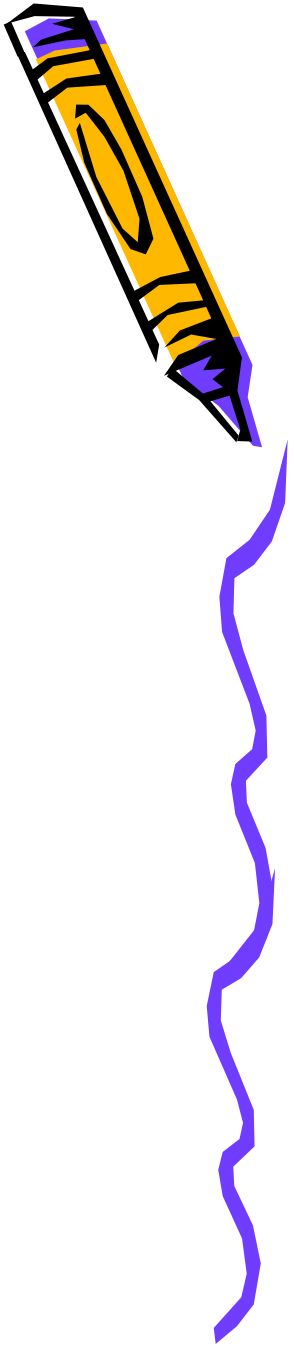


Setting up neighbor array by new rules.

```
for (int x = 0; x < V; x++) {
 switch (x % 4) {
 case 0:
    if (x-n >= 0) neighbor[x][0] = x-n; // N
    if (x+n < V) neighbor[x][1] = x+n;   // S
    if ((x-n >= 0) && (x % n > 0))
        neighbor[x][2] = x-n+1; // NW
    if ((x-n >= 0) && (x % n < n-1))
        neighbor[x][3] = x+n-1; // NE
 ...etc.
}}
```

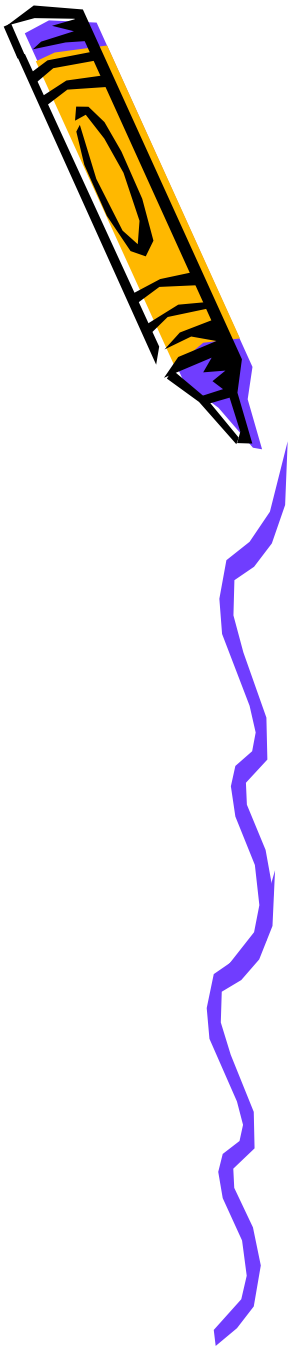


Run program again



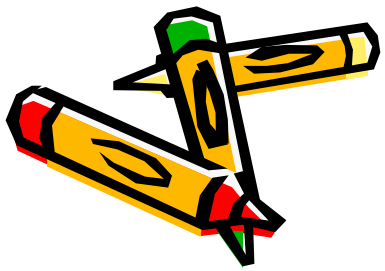
- Works OK on sample data.
- We submit it to judge.
- It is reported as failure.
- After contest, we get judge's data, and retest.
- One of judge's thee data sets seems broken.
  (In contest, you could never have found this out.)

input:
12 33 120
our output:
33 44 31 30 41 52 39 38 49 60 72 84 96 108 120
judges output:
33 34 47 60 72 84 96 108 120



0	1	2	3	4	5	6	7	8	9	10	11
12	13	14	15	16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31	32	33	34	35
36	37	38	39	40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55	56	57	58	59
60	61	62	63	64	65	66	67	68	69	70	71
72	73	74	75	76	77	78	79	80	81	82	83
84	85	86	87	88	89	90	91	92	93	94	95
96	97	98	99	100	101	102	103	104	105	106	107
108	109	110	111	112	113	114	115	116	117	118	119
120	121	122	123	124	125	126	127	128	129	130	131
132	133	134	135	136	137	138	139	140	141	142	143

???



What have we learned?

- Dijkstra's algorithm
- Use of java.util.PriorityQueue
- Subtlety of insert+delete
- Judges sometimes make mistakes; it can be our bad luck if we spend too much time on one problem.
  (I could not have gone through all this analysis during a contest time frame.)

Full program:

www.cs.fsu.edu/~baker/pc/city/City.java