Johnson's Algorithm and the A* Algorithm

Lawrence L. Larmore UNLV

Equivalent Weightings of Directed Graphs

Suppose $G = (V, E)$ is a directed graph. Let W_1 and W_2 be two edge weightings on E, neither weighting allowing a negative cycle. If σ is any path in G, we let $|\sigma|_i$ be the sum of the edges of σ, using the weight W_i. If x, y are nodes of G, we define $d_i(x, y)$ to be the the smallest value of $|\sigma|_i$ for any path σ from x to y.

We say that W_1 and W_2 are *equivalent* if there is some function h such that

$$W_2(x, y) = W_1(x, y) - h(x) + h(y)$$

for all $(x, y) \in E$. It follows that $d_2(x, y) = d_1(x, y) - h(x) - h(y)$ for all $(x, y) \in V \times V$. The matrix *back* remains unchanged.

Johnson's Algorithm

In Johnson's algorithm, given a weighted directed graph (G, W_1) with no negative cycles, we first define G^* to be the *augmented* graph consisting of all the nodes and edges of G, together with one additional node s^*, together with an edge from s^* to every node of G. We then extend the weighting W_1 to G^* by defining $W_1(s^*, x) = 0$ for all $x \in G$.

We now apply the Bellman-Ford algorithm to solve the single source problem on $G*$, where s^* is the source. Let $f(x)$ be the length of the shortest path from s^* to x in G^*. Note that $f(x) \leq 0$ for all x. Use $h(x) = -f(x)$ to define an equivalent weight function W_2.

Claim: All values of W_2 are non-negative.

Proof of Claim: Let $(x, y) \in E$. Then $f(y) \leq f(x) + W_1(x, y)$, since $f(y)$ is the shortest distance from s^* to y, *i.e.,* $0 \leq f(x) - f(y) + W_1(x, y)$. Thus $W_2(x, y) = W_1(x, y) + f(x) - f(y) \geq 0$.

Since (V, E, W_2) is a weighted directed graph with no negative edges, we can apply Dijkstra's algorithm n times. The solution to the all-pairs problem on (V, E, W_2) is simply the combination of the n solutions to the single source problem, where each node is chosen once to be the source. Our output consists of arrays $\{V_2(u, v)\}$ and $\{back(u, v)\}$ For all $(u, v) \in V \times V$. However, we would like the values for the original problem, where the edge weights are given by W_1. We can create the final matrix $\{V_1(u, v)\}$ by setting $V_1(u, v) = V_2(u, v) - f(x) + f(y)$.

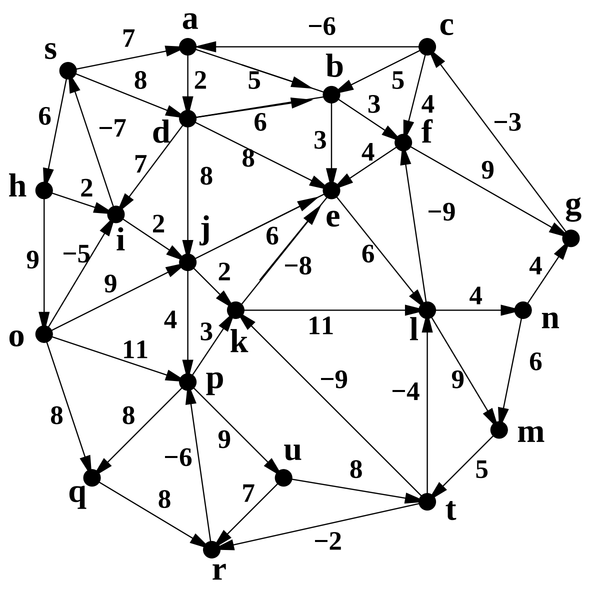
Figure 1: G, a weighted directed graph.

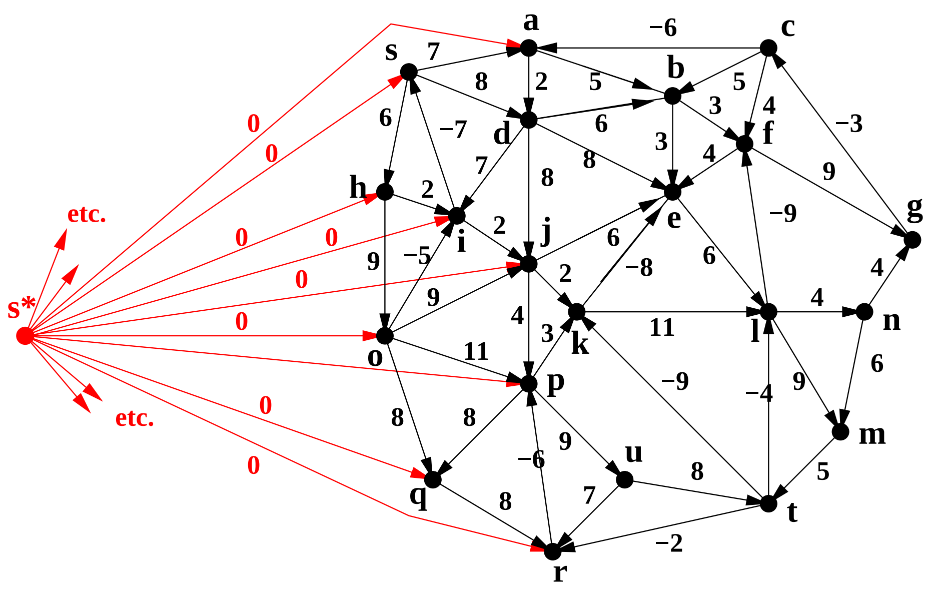
Figure 2: The augmented graph G^*. Not all the augmentation edges are shown.

Exercise

Fill in the values of f in Figure 3(a), then fill in the values of W_2 in Figure 3(b).

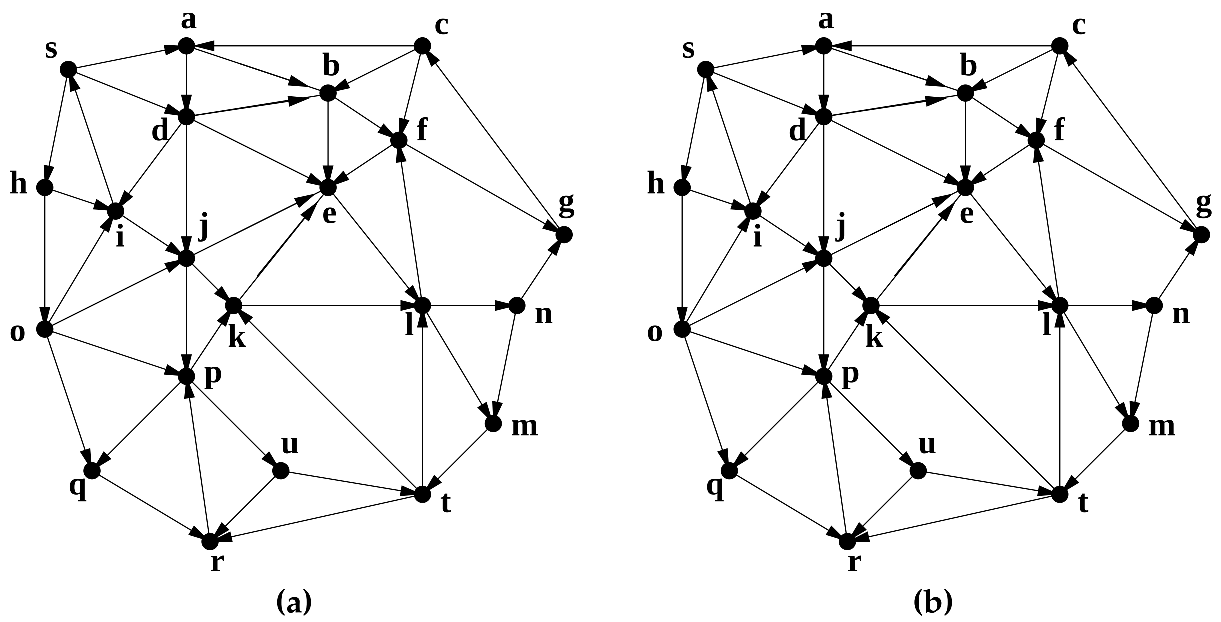

Figure 3

The A* Algorithm

The A* algorithm solves the single pair shortest path problem fast under some circumstances. We assume that G is a directed graph with a weight function W_1, such that there are no negative cycles, and that s and t are distinguished *source* and *target* nodes. We also assume that we are given a heuristic h on V condition that $h(x) \leq W_1(x,y) + h(y)$ for any edge (x,y) of G. Let $W_2(x,y) = W_1(x,y) - h(x) + h(y)$. We can then solve the single pair shortest problem in G_1 by solving the same problem for (V, E, G_2).

Will it be faster? That depends on the heuristic. Remember: there is no general way to find a heuristic; it has to come out of the specific application in some way, or be given to you by the person (me in this case) who gives you the problem.

Exercise

Consider the graph G illustrated below. Figure 4(a) shows G has a weighted graph, where the number on an edge $\{x,y\}$ is equal to both $W_1(x,y)$ and $W_1(y,x)$. That is, we can think of G_1 as a *symmetric* weighted graph. How much time would it take to find the shortest path from s to t? Your heap would eventually contain all the nodes. (Or maybe all but one; I'm not sure.) So, you pay for the single source solution, even though you don't need it.

Figure 4(b) shows the heuristic, in red. (Don't ask how I came up with it. I just made up numbers that work.)

1. Mark the Figure **??** with the second weight function, W_2.
2. Use Dijkstra's algorithm for $G_2 = (G, W_2)$ to find the shortest path, in G_1, from s to t. It is amazingly fast.

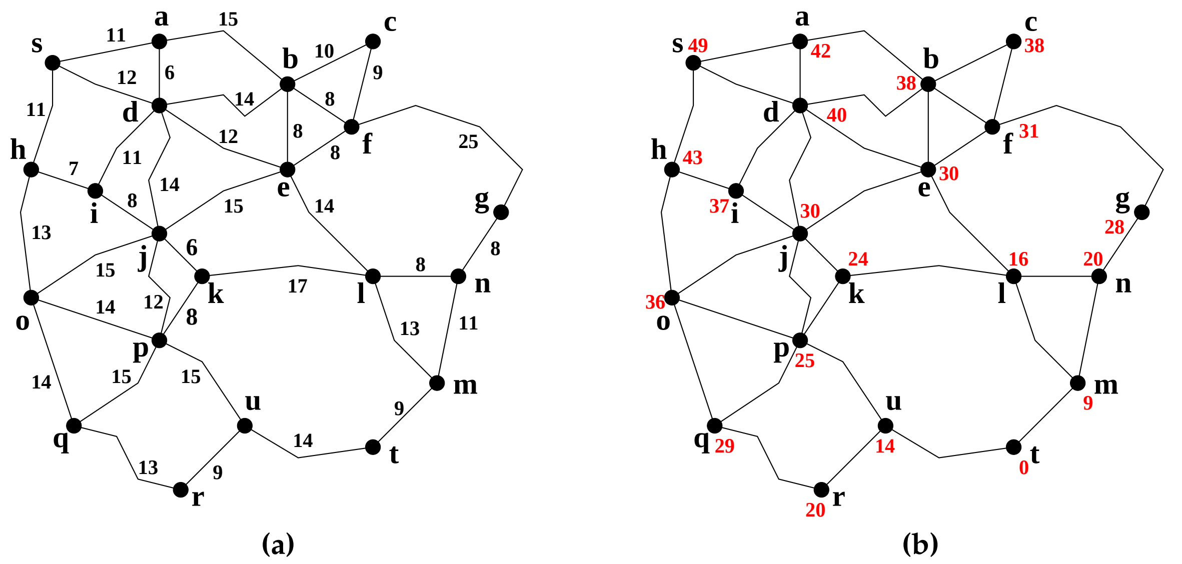

Figure 4

Here the exercise. Fill in the adjusted edge weights in Figure 5. Note that each edge has been replaced by two Edges, one in each direction. When you now run Dijkstra's algorithm, which nodes will be left on the heap? Which nodes will never even be inserted into the heap?

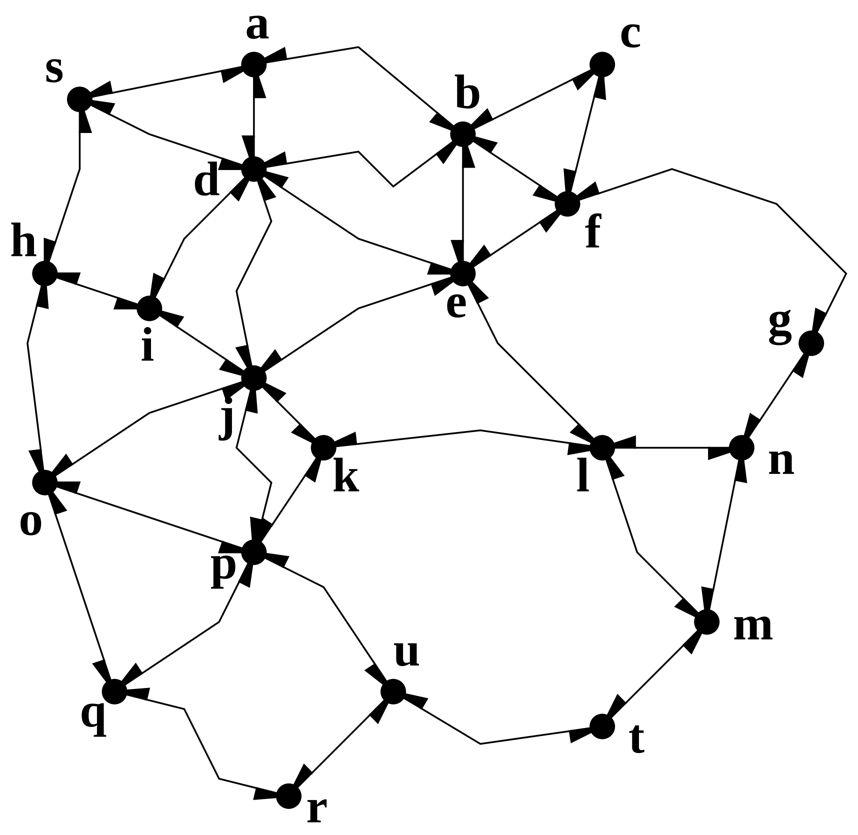

Figure 5

Edit Distance

The dynamic programming algorithm for edit distance can be replaced by Dijkstra's algorithm. The running is then much smaller, if the two words are close.

Find the edit distance between *fluorouracil* and *florouricil*. The matrix obtained by dynamic programming is shown in Figure 6. Figure 7 shows values at those nodes which will be visited and placed in the heap when we use Dijkstra's algorithm.

		f	*l*	*o*	*r*	*o*	*u*	*r*	*i*	*c*	*i*	*l*
	0	1	2	3	4	5	6	7	8	9	10	11
f	1	**0**	1	2	3	4	5	6	7	8	9	10
l	2	1	**0**	1	2	3	4	5	6	7	8	9
u	3	2	**1**	1	2	3	4	5	6	7	8	9
o	4	3	2	**1**	2	2	3	4	5	6	7	8
r	5	4	3	2	**1**	2	3	4	5	6	7	8
o	6	5	4	3	2	**1**	2	3	4	5	6	7
u	7	6	5	4	3	2	**1**	2	3	4	5	6
r	8	7	6	5	4	3	2	**1**	2	3	4	5
a	9	8	7	6	5	4	3	2	**2**	3	4	5
c	10	9	8	7	6	5	4	3	3	**2**	3	4
i	11	10	9	8	7	6	5	4	4	3	**2**	3
l	12	11	10	9	8	7	6	5	5	4	3	**2**

Figure 6

		f	*l*	*o*	*r*	*o*	*u*	*r*	*i*	*c*	*i*	*l*
	0	1	2	3								
f	1	**0**	1	2	3							
l	2	1	**0**	1	2	3						
u	3	2	**1**	1	2	3						
o		3	2	**1**	2	2	3					
r			3	2	**1**	2	3					
o				3	2	**1**	2	3				
u					3	2	**1**	2	3			
r						3	2	**1**	2	3		
a							3	2	**2**	3		
c								3	3	**2**	3	
i										3	**2**	3
l											3	**2**

Figure 7

Figure 8 shows the heuristic h obtained by considering only the length of the remaining suffix. The adjusted edit distance between the two words is now only 1. When we use Dijkstra's algorithm using adjusted weights, we only compute the values shown in Figure 9.

		f	*l*	*o*	*r*	*o*	*u*	*r*	*i*	*c*	*i*	*l*
	1	2	3	4	5	6	7	8	9	10	11	12
f	0	1	2	3	4	5	6	7	8	9	10	11
l	1	0	1	2	3	4	5	6	7	8	9	10
u	2	1	0	1	2	3	4	5	6	7	8	9
o	3	2	1	0	1	2	3	4	5	6	7	8
r	4	3	2	1	0	1	2	3	4	5	6	7
o	5	4	3	2	1	0	1	2	3	4	5	6
u	6	5	4	3	2	1	0	1	2	3	4	5
r	7	6	5	4	3	2	1	0	1	2	3	4
a	8	7	6	5	4	3	2	1	0	1	2	3
c	9	8	7	6	5	4	3	2	1	0	1	2
i	10	9	8	7	6	5	4	3	2	1	0	1
l	11	10	9	8	7	6	5	4	3	2	1	0

Figure 8

		f	*l*	*o*	*r*	*o*	*u*	*r*	*i*	*c*	*i*	*l*
	0	2										
f	0	**0**	2									
l	2	0	**0**	2								
u		2	**0**	1	3							
o			2	**0**	2							
r				2	**0**	2						
o					2	**0**	2					
u						2	**0**	2				
r							2	**0**	2			
a								2	**1**	3		
c									3	**1**	3	
i										3	**1**	3
l											3	**1**

Figure 9

Find the edit distance between ”Levenshtein” and ”Levinstine.”