My implementation of the hillclimb algorithm is as follows:

```
def hillclimb(s0,goaltest,h):
    s = s0
    iterations = 0
    while not goaltest(s):
        iterations = iterations + 1
        weights = []
        s2 = []
        for a, k in s.successors():
            cost = a.cost+h(k)
            weights.append(100/(cost**9))
            s2.append(k)
        s = random.choices(s2, weights=weights, k=1)
        s = s[0]
```

I tried out couple different weight scenarios. In the end, I decided to approach the problem as suggested in the task description. I wanted to put more emphasis on choosing lowest f-value state. Hence the aggressive multiplication of the cost to the power of 9. I found that in the first few tests raising to the power of 9 seemed to be most efficient. I experimented in the region of 1-11. I'll provide a table of the runs of algorithms on the next page. Runtimes seemed to be all in the same region, so I decided to choose values from one of the runs, since they quite realistically represented all results simultaneously.

data in columns as follows: runtime // cost/optimal cost						
	A*	W2A*	W1.5A*	W5A*	hillclimbing	IDA*
example 0	0.025744 // 6/6	0.000894 // 6/6	0.00928 // 6/6	0.00087 // 6/6	0.00356 // 6/6	0.002656 // 6/6
1a	0.009278 // 8/8	0.002292 // 8/8	0.003622 // 8/8	0.001014 // 9/8	0.003417 // 8/8	0.004786 // 8/8
1b	0.00105 // 6/6	0.000965 // 6/6	0.000985 // 6/6	0.000973 // 6/6	0.0375367 // 15/6	0.000839 // 6/6
1c	0.003625 // 7/7	0.003406 // 7/7	0.003409 // 7/7	0.0034049 // 7/7	0.0439029 // 17/7	0.001261 // 7/7
1d	0.013099 // 11/11	0.0086270 // 11/11	0.0086410 // 11/11	0.008768 // 11/11	0.274233 // 26/11	0.047914 // 11/11
1e	0.284821 // 12/12	3.28713 // 16/12	0.299312 // 13/12	3.274565 // 16/12	0.224946 // 31/12	0.105441 // 12/12
1f	0.283484 // 13/13	0.07416 // 14/13	0.038037 // 13/13	0.073978 // 14/13	3.255977 // 22/13	0.056588 // 13/13
1g	6.106988 // 15/15	0.0725440 // 16/15	0.0388349 // 15/15	0.072317 // 16/15	242.357568 // 29/15	0.724915 // 15/15
1h	3.751827 // 15/15	0.400103 // 19/15	0.09595 // 16/15	0.385038 // 19/15	4.905566 // 30/15	1.451567 // 15/15
1i	14.305309 // 17/17	0.040610 // 18/17	0.297573 // 17/17	0.0407480 // 18/17	28.244593 // 25/17	7.5175740 // 17/17
1j	32.523705 // 15/15	2.30906 // 19/15	0.272632 // 16/15	2.32685 // 19/15	54.63520 // 33/15	5.6221 // 15/15
1k	49.72733 // 19/19	0.06716 // 20/19	0.076969 // 20/19	0.06874 // 20/19	14.6119969 // 29/19	17.56977 // 19/19
1l	75.3657 // 18/18	0.331578 // 22/18	0.424043 // 20/18	0.330837 // 22/18	119.86075 // 45/18	22.979726 // 18/18
1m	21.014295 // 18/18	0.020867 // 18/18	0.020777 // 18/18	0.0207449 // 18/18	93.287662 // 57/18	29.277727 // 18/18
example 2	8.854649 // 16/16	0.051491 // 20/16	0.114569 // 16/16	0.05169 // 20/16	7.4671512 // 32/16	9.900077 // 16/16
example 3a	0.000421 // 6/6	0.000228 // 6/6	0.000231 // 6/6	0.000226 // 6/6	0.0009489 // 6/6	0.0006859 // 6/6
example 3b	1.4304759 // 18/18	0.027183 // 18/18	0.07975 // 18/18	0.025033 // 18/18	0.806879 // 42/18	No plan
example 3c	159.058045 // 26/26	0.8902669 // 28/26	2.124839 // 26/26	0.787289 // 28/26	6.953372 // 98/26	No plan
example 4a	0.1606600 // 23/23	0.044631 // 23/23	0.05543 // 23/23	0.044826 // 23/23	0.008556 // 33/23	No plan
example 4b	3.343814 // 30/30	0.853484 // 38/30	1.391117 // 30/30	0.8605210 // 38/30	dnf	dnf