8 Hill Climbing

Hill climbing is a technique for certain classes of optimization problems. The idea is to start with a sub-optimal solution to a problem (i.e., *start at the base of a hill*) and then repeatedly improve the solution (*walk up the hill*) until some condition is maximized (*the top of the hill is reached*).

Hill-Climbing Methodology
Construct a sub-optimal solution that meets the constraints of the problem# Take the solution and make an improvement upon it# Repeatedly improve the solution until no more improvements are necessary/possible

One of the most popular hill-climbing problems is the network flow problem. Although network flow may sound somewhat specific it is important because it has high expressive power: for example, many algorithmic problems encountered in practice can actually be considered special cases of network flow. After covering a simple example of the hill-climbing approach for a numerical problem we cover network flow and then present examples of applications of network flow.

8.1 Newton's Root Finding Method

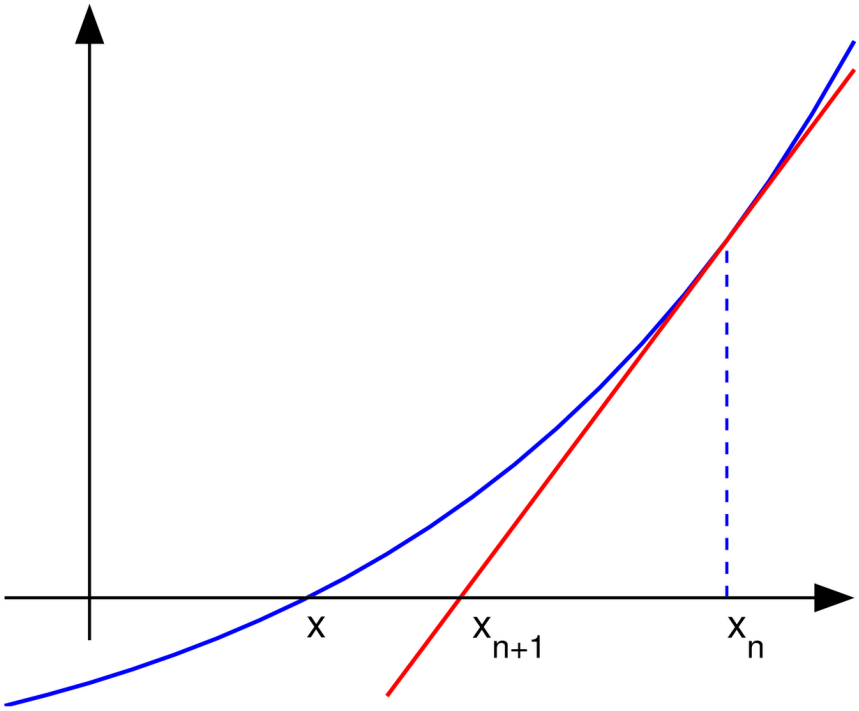

Figure 9 An illustration of Newton's method: The zero of the *f(x)* function is at x. We see that the guess x_{n+1} is a better guess than x_n because it is closer to x. (*from Wikipedia*[a])

a http://en.wikipedia.org/wiki/Newton%27s%20method

Newton's Root Finding Method is a three-centuries-old algorithm for finding numerical approximations to roots of a function (that is a point x where the function $f(x)$ becomes zero), starting from an initial guess. You need to know the function $f(x)$ and its first derivative $f'(x)$ for this algorithm. The idea is the following: In the vicinity of the initial guess x_0 we can form the Taylor expansion of the function

$$f(x) = f(x_0 + \epsilon) \approx f(x_0) + \epsilon f'(x_0) + \frac{\epsilon^2}{2} f''(x_0) + \ldots$$

which gives a good approximation to the function near x_0. Taking only the first two terms on the right hand side, setting them equal to zero, and solving for ϵ, we obtain

$$\epsilon = -\frac{f(x_0)}{f'(x_0)}$$

which we can use to construct a better solution

$$x_1 = x_0 + \epsilon = x_0 - \frac{f(x_0)}{f'(x_0)}.$$

This new solution can be the starting point for applying the same procedure again. Thus, in general a better approximation can be constructed by repeatedly applying

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$

As shown in the illustration, this is nothing else but the construction of the zero from the tangent at the initial guessing point. In general, Newton's root finding method converges quadratically, except when the first derivative of the solution $f'(x) = 0$ vanishes at the root.

Coming back to the "Hill climbing" analogy, we could apply Newton's root finding method not to the function $f(x)$, but to its first derivative $f'(x)$, that is look for x such that $f'(x) = 0$. This would give the extremal positions of the function, its maxima and minima. Starting Newton's method close enough to a maximum this way, we climb the hill.

Instead of regarding continuous functions, the hill-climbing method can also be applied to discrete networks.

8.2 Network Flow

Suppose you have a directed graph (possibly with cycles) with one vertex labeled as the source and another vertex labeled as the destination or the "sink". The source vertex only has edges coming out of it, with no edges going into it. Similarly, the destination vertex only has edges going into it, with no edges coming out of it. We can assume that the graph fully connected with no dead-ends; i.e., for every vertex (except the source and the sink), there is at least one edge going into the vertex and one edge going out of it.

We assign a "capacity" to each edge, and initially we'll consider only integral-valued capacities. The following graph meets our requirements, where "s" is the source and "t" is the destination:

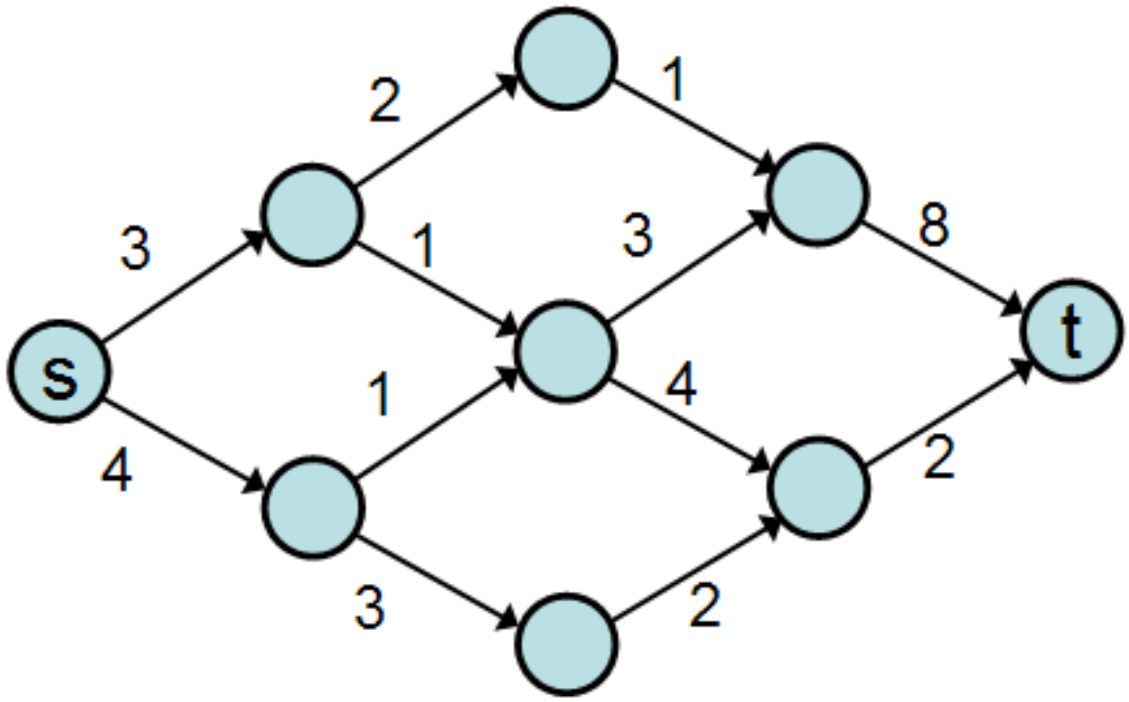

Figure 10

We'd like now to imagine that we have some series of inputs arriving at the source that we want to carry on the edges over to the sink. The number of units we can send on an edge at a time must be less than or equal to the edge's capacity. You can think of the vertices as cities and the edges as roads between the cities and we want to send as many cars from the source city to the destination city as possible. The constraint is that we cannot send more cars down a road than its capacity can handle.

The goal of network flow is to send as much traffic from s to t as each street can bear.

To organize the traffic routes, we can build a list of different paths from city s to city t. Each path has a carrying capacity equal to the smallest capacity value for any edge on the path; for example, consider the following path p:

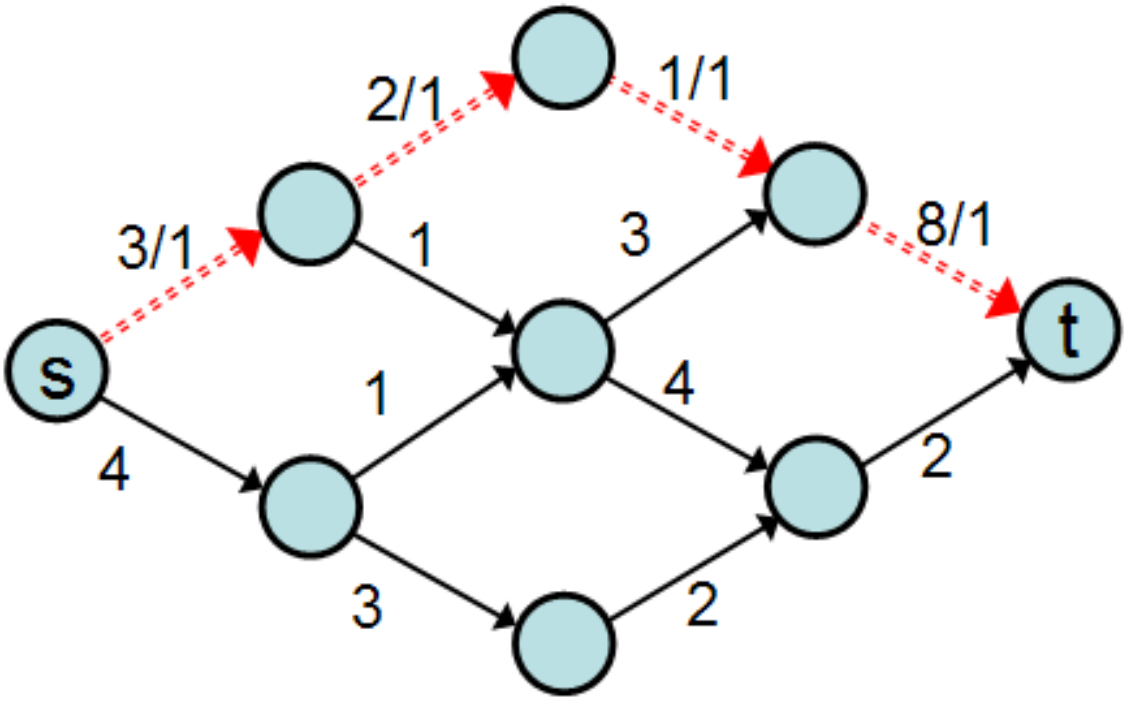

Figure 11

Even though the final edge of p has a capacity of 8, that edge only has one car traveling on it because the edge before it only has a capacity of 1 (thus, that edge is at full capacity). After using this path, we can compute the **residual graph** by subtracting 1 from the capacity of each edge:

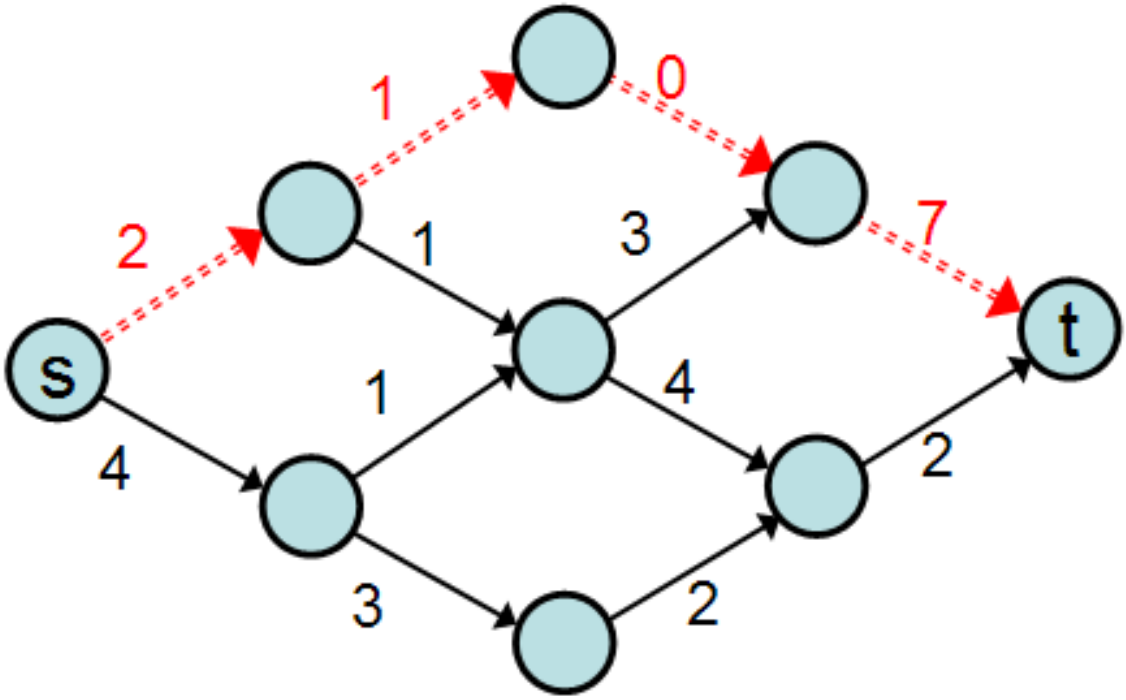

Figure 12

(We subtracted 1 from the capacity of each edge in p because 1 was the carrying capacity of p.) We can say that path p has a flow of 1. Formally, a **flow** is an assignment $f(e)$ of values to the set of edges in the graph $G = (V, E)$ such that:

1. $\forall e \in E : f(e) \in \mathbb{R}$
2. $\forall (u, v) \in E : f((u, v)) = -f((v, u))$
3. $\forall u \in V, u \neq s, t : \sum_{v \in V} f(u, v) = 0$
4. $\forall e \in E : f(e) \leq c(e)$

Where s is the source node and t is the sink node, and $c(e) \geq 0$ is the capacity of edge e. We define the value of a flow f to be:

$$\text{Value}(f) = \sum_{v \in V} f((s, v))$$

The goal of network flow is to find an f such that Value(f) is maximal. To be maximal means that there is no other flow assignment that obeys the constraints 1-4 that would have a higher value. The traffic example can describe what the four flow constraints mean:

1. $\forall e \in E : f(e) \in \mathbb{R}$. This rule simply defines a flow to be a function from edges in the graph to real numbers. The function is defined for every edge in the graph. You could also consider the "function" to simply be a mapping: Every edge can be an index into an array and the value of the array at an edge is the value of the flow function at that edge.

2. $\forall (u,v) \in E : f((u,v)) = -f((v,u))$. This rule says that if there is some traffic flowing from node u to node v then there should be considered negative that amount flowing from v to u. For example, if two cars are flowing from city u to city v, then negative two cars are going in the other direction. Similarly, if three cars are going from city u to city v and two cars are going city v to city u then the net effect is the same as if one car was going from city u to city v and no cars are going from city v to city u.
3. $\forall u \in V, u \neq s, t : \sum_{v \in V} f(u,v) = 0$. This rule says that the net flow (except for the source and the destination) should be neutral. That is, you won't ever have more cars going into a city than you would have coming out of the city. New cars can only come from the source, and cars can only be stored in the destination. Similarly, whatever flows out of s must eventually flow into t. Note that if a city has three cars coming into it, it could send two cars to one city and the remaining car to a different city. Also, a city might have cars coming into it from multiple sources (although all are ultimately from city s).
4. $\forall e \in E : f(e) \leq c(e)$.

8.3 The Ford-Fulkerson Algorithm

The following algorithm computes the maximal flow for a given graph with non-negative capacities. What the algorithm does can be easy to understand, but it's non-trivial to show that it terminates and provides an optimal solution.

```
function net-flow(graph (V, E), node s, node t, cost c): flow
  initialize f(e) := 0 for all e in E
  loop while not done
    for all e in E:                          // compute residual capacities
      let cf(e) := c(e) - f(e)
    repeat

    let Gf := (V, {e : e in E and cf(e) > 0})

    find a path p from s to t in Gf          // e.g., use depth first search
    if no path p exists: signal done

    let path-capacities := map(p, cf)        // a path is a set of edges
    let m := min-val-of(path-capacities)     // smallest residual capacity of p
    for all (u, v) in p:                     // maintain flow constraints
      f((u, v)) := f((u, v)) + m
      f((v, u)) := f((v, u)) - m
    repeat
  repeat
end
```

8.4 Applications of Network Flow

1. finding out maximum bi - partite matching . 2. finding out min cut of a graph .