Introduction to Width Independent MWU

네트워크 플로우는 효율적인 그래프 알고리즘을 고안하기 위한 기초적인 도구이다. 이미 글을 통해서 여러 번 밝힌 바가 있지만 이 분야에 대해서 수도 없이 많은 연구들이 진행되었고, 최근 Almost-Linear Time Minimum Cost Flow 가 가능하다는 사실이 알려져서 많은 화제를 모은 바도 있다.

이론적으로는 효율적인 네트워크 플로우 알고리즘에 대한 연구가 상당히 진전이 되어 있지만, 정확하게 네트워크 플로우 문제를 해결하려는 경우는 아직 Push-relabel보다 실용적인 방법이 잘 알려져 있지 않다. 이론적으로 효율적인 플로우 알고리즘은, 그 알고리즘들의 실제 수행 시간이 어떤가와 별개로 아직 해당 알고리즘들을 잘 구현한 구현체 자체가 존재하지 않아서 효율성에 대한 논의가 아직은 성급하다.

이와 별개로 근사적으로(*approximate*) 네트워크 플로우 문제를 해결하는 경우에 대해서는 굉장히 효율적인 알고리즘들이 존재하는데, 그러한 것들이 지금 이 글에서 논의하게 될 Width-Independent MWU 알고리즘들이다. 이 알고리즘들은 Near-Linear Time ($O(n \log^c(n))$) 에 동작하여, 최근 연구에서 등장한 Almost-Linear Time ($O(n^{1+o(1)})$) 알고리즘들보다 효율적이다.

이러한 알고리즘은 여러 장점을 가지는데, 일단 첫 번째로는 실용적인 구현이 가능할 정도로 알고리즘이 충분히 간단하다. 특히 알고리즘이 *approximate* 하다는 점이 아주 좋은 장점인데, 실용성 면에서는 Exact한 최대 유량을 굳이 찾을 필요가 없기 때문이다. 두 번째로, 알고리즘 자체의 Flexibility가 아주 좋아서 플로우의 형태가 아닌 더 일반적인 LP의 형태에서도 작동하는 경우가 많다. 플로우는 LP의 특수 경우라고 볼 수 있는데, 플로우와 LP 경계 정도에 있는 문제는 이러한 Width-Independent MWU로도 충분히 해결이 가능하다. 특히 이 글에서는 Garg-Konemann을 통해서 최대 유량 문제와 Packing LP를 푸는 방법을 살펴보고 두 방법의 유사성에 대해서 짚어볼 것이다.

이 글에서는 Width-independent MWU의 두 가지 예시인, Garg-Konemann과 Kent의 알고리즘을 돌아보면서 이러한 테크닉의 기본 설계를 익히는 것을 목표로 한다.

1. Garg-Konemann Algorithm for Multi-Commodity flow

Garg-Konemann은 다음과 같은 Primal Problem을 해결하려고 한다. 이 문제를 **Multi-Commodity Flow** 라고 한다.

Primal: Edge capacity $c : E \to R^+$ 가 있는 그래프에 k 개의 source-sink pair (s_i, t_i) 가 있다. 이 pair로 i 번째 commodity를 f_i 만큼 수송했다고 할때, Capacity 조건을 만족하면서 $\sum f_i$ 최대화

이는 일반적인 Maximum Flow 문제의 일반화로, Maximum flow는 $k = 1$ 인 특수 경우에 대응된다.

Dual: 그래프의 간선에 가중치 l 을 주는데, $\sum_e l(e)c(e)$ 를 최소화해야 하고, 조건은 (s_i, t_i) 를 잇는 최단 경로가 모두 1 이상이어야 함. 달리 말해, $\beta = \frac{\sum_e l(e)c(e)}{\min_i dist_l(s_i,t_i)}$ 최소화.

Algorithm

$l_i(e)$ 를 i 번째 iteration 후 간선의 길이라고 하고 $f_i(e)$ 를 i 번째 iteration 후 flow라고 하자. 즉 첫 번째는 dual value, 두 번째는 primal value이다. 이 두 값을 같이 관리한다.

Garg-Konemann 알고리즘은 어떤 경로에 플로우를 흘려 줄 때, 해당 경로의 간선 길이를 흘려준 플로우 양에 비례하게 증가시키는 것을 반복한다. 이 연산을 할 경우 플로우가 많이 흐르는 간선은 가중치가 크다는 상관관계를 유지시킬 수 있다. 만약에 이러한 그래프에서 플로우를 최단 경로로 흘려 줄 때, 플로우는 이미 안 흘린 곳으로 흐르려는 경향성을 보일 것이다. 이를 통해서 각 간선의 플로우양을 "공평하게" 맞춰줄 수 있고, 간선을 효율적으로 포화시키는 것이 목표이다.

구체적으로 논의하면, $dist_l(s_i, e_i)$ 를 최소화하는 source-sink path P 를 고른 후 P로 flow를 흘려준다. 이 때 흘리는 flow는 P 상 capacity의 최솟값이고, 이 값을 c 라고 하자.

초기 $l_0(e) = \delta$, $f_0(e) = 0$ 로 설정했을 때, 알고리즘은 매번

- $e \in P$ 에 대해서 $f_i(e) = f_{i-1}(e) + c$ 로 설정.
- $e \in P$ 에 대해서 $l_i(e) = l_{i-1}(e)(1 + \frac{\epsilon c}{c(e)})$ 로 설정.

이 과정을, 모든 source-sink pair의 최단 경로가 1 이상이 될 때까지 반복하는 것이 Garg-Konemann Algorithm이다. Dijkstra가 $O(m \log m)$ 이라고 하고, 반복 횟수가 T 라고 하면, 이 알고리즘의 시간 복잡도는 $O(Tkm \log m)$ 이다. 이제 이 알고리즘이 충분히 많이 반복되었을 때 올바름을 증명한다.

Proof

$D(i) = \sum_e l_i(e)c(e)$ 라 하고, $\alpha(i) = \min_j dist_{l_i}(s_j, t_j)$ 라고 하자. $f_i = \sum_e f_i(e)$ 라고 하자. i 번째 iteration 후 Primal의 답은 $\sum_e f_i(e)$ 고 Dual의 답은 $\frac{D(i)}{\alpha(i)}$ 이다.

L 을 Flow graph 상에서의 최장 경로의 길이라고 하자. 여기서 경로의 길이는 간선 개수를 뜻한다. 상수 ϵ 에 대해서 $\lceil \frac{1}{\epsilon} \log_{1+\epsilon} L \rceil$ 번 iteration한 결과를 분석하자.

$D(i) = \sum_e l_i(e)c(e) = \sum_e l_{i-1}(e)c(e) + \epsilon \sum_e l_{i-1}(e)c = D(i-1) + \epsilon\alpha(i-1)c = D(i-1) + \epsilon\alpha(i-1)(f_i - f_{i-1})$

고로

$D(i) = D(0) + \epsilon \sum_{j \in [i]} \alpha(j-1)(f_j - f_{j-1})$

이제 length function $l_i - l_0$ 을 생각해 보자 (pairwise subtraction).

$D(l_i - l_0) = D(i) - D(0)$

$\alpha(l_i - l_0) \geq \alpha(i) - \delta L$

이 성립한다.

$l_i - l_0$ 역시 valid한 length function이니, β 의 정의에 의해

$\beta \leq \frac{D(l_i - l_0)}{\alpha(l_i - l_0)} \leq \frac{D(i) - D(0)}{\alpha(i) - \delta L}$

전개하면

$\beta(\alpha(i) - \delta L) \leq \epsilon \sum_{j \in [i]} \alpha(j-1)(f_j - f_{j-1})$

$\alpha(i) \leq \delta L + \frac{\epsilon}{\beta} \sum_{j \in [i]} \alpha(j-1)(f_j - f_{j-1})$

$\alpha(i) \leq \alpha(i-1)(1 + \frac{\epsilon}{\beta}(f_i - f_{i-1}))$

$\alpha(i) \leq \alpha(i-1)e^{\frac{\epsilon}{\beta}(f_i - f_{i-1})}$

$\alpha(i) \leq \alpha(0)e^{\frac{\epsilon f_i}{\beta}}$

$\alpha(i) \leq \delta L e^{\frac{\epsilon f_i}{\beta}}$ $(\alpha(0) \leq \delta L)$

$1 \leq \delta L e^{\frac{\epsilon f_i}{\beta}}$ (종료 조건에 의해 $\alpha(i) \geq 1$)

$0 \leq \log(\delta L) + \frac{\epsilon f_i}{\beta}$

$\frac{-\log(\delta L)}{\epsilon} \leq \frac{f_i}{\beta}$

$\frac{\beta}{f_i} \leq \frac{\epsilon}{-\log(\delta L)}$

이 된다. 이제 Main Claim을 소개한다.

Claim. $\frac{f_i}{\log_{1+\epsilon}\frac{1+\epsilon}{\delta}}$ 의 값을 가지는 플로우가 존재한다.

Proof. 간선 e 를 생각하자. 매번 $c(e)$의 플로우를 e를 통해 흘릴 때, e 의 길이는 $1+\epsilon$ 배 이상 증가한다. 마지막으로 증가할 때 e 는 길이가 1 미만인 경로 위에 있는 간선이었다. 간선의 길이는 $1+\epsilon$ 배 이하로 증가하니, $l_t(e) < 1+\epsilon$ 이다. $l_0(e) = \delta$ 이니, e 를 통해 흐르는 유량은 최대 $c(e)\log_{1+\epsilon}\frac{1+\epsilon}{\delta}$ 이다. 이를 $\log_{1+\epsilon}\frac{1+\epsilon}{\delta}$ 으로 나누면 알고리즘이 주장하는 양의 플로우를 얻을 수 있다. ■

한편, Primal solution과 Dual solution의 값의 비율은 $\frac{\beta}{f_i}\log_{1+\epsilon}\frac{1+\epsilon}{\delta}$ 이다. 이를 위에서 얻은 부등식과 조합하면 $\frac{\epsilon}{-\log(\delta L)}\log_{1+\epsilon}\frac{1+\epsilon}{\delta}$ 이다. 만약 $\delta = (1+\epsilon)((1+\epsilon)L)^{-\frac{1}{\epsilon}}$ 으로 대입할 경우, Primal과 Dual의 비율이 $\leq (1-\epsilon)^{-2}$ 가 됨을 식 전개로 증명할 수 있다. Exact하게 풀었을 때 두 값은 같으니 이것이 우리의 오차율이고, Application에 따라서 적당히 Epsilon을 조정해 주면 된다.

2. Garg-Konemann Algorithm for Packing LP

이제 Garg-Konemann Algorithm을 사용하여 Packing LP를 해결한다. Packing LP의 Primal problem은 일반적인 LP와 거의 동일하다.

Primal. $max(c^Tx)$ subject to $Ax \leq b, x \geq 0$

차이점이 딱 한 가지 있는데, 바로 A, b, c 의 **모든 항이** 0 이상의 수라는 것이다. 이 LP의 Dual은 당연히

Dual. $min(b^Ty)$ sjt $A^Ty \geq c, y \geq 0$

이 되고 여기서도 당연히 A, b, c 의 **모든 항이** 0 이상의 수이다.

Algorithm

각 Row를 *간선* 으로 보자. i 번 row의 *용량* 은 $b(i)$ 이다.

x_i 컴포넌트를 1 증가시킬 경우 $c(i)$의 이득을 얻고, 각 간선 j 의 용량을 $A_{j,i}$ 만큼 소모한다. 이 때 각 간선의 용량은 $b(i)$ 이하여야 한다. 간선들을 순서대로 나열하면, 이는 Path에 대해서 유량을 흘려준 것이라고 상상할 수 있다. 즉, 각 Column은 N 개의 간선을 가지는 path라고 생각할 수 있다. 각 변수에 대응되는 간선 집합은 동일하지만, 용량을 얼마나 사용하는가에 차이가 있다.

이제 Dual variable y 는 간선 i 의 길이에 대응된다고 볼 수 있다. j 번 변수가 대응되는 경로의 길이를 $length(j) = \sum_i \frac{A(i,j)y(i)}{c(j)}$ 라고 하면, 목표는 $length(j)$ 가 모두 1 이상이 되게 맞춰주는 것이다. $\alpha(y) = min_j length(j)$, $D(y) = b^Ty$ 라고 하면 우리는 $\frac{D(y)}{\alpha(y)}$ 를 최대화하고 싶다.

이제 "경로, 용량" analogy를 완성했으니 이에 맞춰서 Multi-commodity flow의 프레임워크를 끼워맞춰보자. Multi-commodity flow에서는 매번 경로 길이를 최소화하는 쪽으로 유량을 흘려준 후, 흘린 쪽의 길이를 증가시킨다. 즉

- $length(j)$ 를 최소화하는 j 를 고른다. 이걸 q 라고 하자.
- 해당 column에서 bottleneck이 되는 row p 를 고른다. 다시 말해 $b(i)/A(i,q)$ 를 최소화하는 i 가 p 이다.
- Primal $x(q)$ 를 $b(p)/A(p,q)$ 만큼 늘린다.
- Dual $y_k(i)$ 에 $(1+\epsilon\frac{b(p)/A(p,q)}{b(i)/A(i,q)})$ 를 곱해준다.

그리고 $length(j)$가 모두 1 이상일 때 종료한다.

여기서 Primal $x(p)$ 를 잔여 용량만큼 늘려주는 것이 아니라 잔여 용량과 **무관하게** $b(i)/A(i,q)$ 만큼 올려준다는 점에 유의해야 한다. Multi-commodity flow에서 사용한 테크닉을 그대로 사용하는게 아니라 그냥 "그런 느낌으로" 사용하는 것일 뿐이다. Primal을 잔여 용량만큼 늘리지 않은 관계로 알고리즘이 종료된 후 x 는 Feasible solution이 아닐 수 있다. 이는 x 에 적당한 상수배를 곱해서 해결할 것이다.

Garg-Konemann in MWU framework

위 논의를 따라가면서 Garg-Konemann을 사용한 Packing LP의 해결을 Multi-commodity flow의 framework에서 분석하는 것이 억지라는 느낌을 받았을 것이다. Garg-Konemann를 제대로 분석하기 위해서는 MWU Framework라는 틀로 생각해야 한다. MWU Framework에 대해서는 이전 글에서도 한번 다룬 적이 있는데, 여기서 다시 간단하게 요약한다. (사실 저 글을 쓸 때는 MWU를 Lagrangian의 관점에서 분석하지 못하였다. 여기서 다시 짧게 읽어보는 게 저 글보다 오히려 읽기 더 좋을 수 있겠다.)

Convex space P 에서 $Ax \geq 1$ 를 만족하는 해가 있는지 찾는 문제를 생각해 보자. 이 때 Convex space P 는 *굉장히 쉽고*, 이 안에서는 이런저런 최적화 문제들에 대한 Trivial한 풀이들이 존재한다고 하자. $Ax \geq 1$ 은 어려운 문제지만, 이 점을 활용하여 최대한 P 에 대한 문제로 변환해 보자.

이를 위해 Lagrangian relaxation (Alien's trick?) 의 아이디어를 생각해 볼 수 있다. 우리는 $Ax-1$ 의 각 원소가 0 이상이기를 원한다. $Ax-1$ 을 어떠한 positive vector y 랑 내적한 값, $y(Ax-1)$ 을 생각해 보면, 이 값이 크면 클수록 해에 더 가깝다고 생각할 수 있다. 고로 절묘한 y 를 구한 후 Convex space P 에서 $y(Ax-1)$ 을 최대화하는, 다른 말로 yAx 를 최대화하는 문제를 해결하면, 이 문제를 해결하는데 도움이 될 것이다. 이 문제를 해결해 주는 것을 **Oracle** 이라 부른다.

이것이 abstract한 idea이고, 이 idea를 구체화하는 데 MWU Framework가 사용된다. MWU Framework에서 각 부등식에는 *전문가* i 가 assign된다. 이 전문가의 *신뢰도* 가, 바로 여기서 사용할 절묘한 y 가 되고, 초기 이 값은 모두 1 이다 ($y_i = 1$). 매 MWU Step에서

- 전문가들의 신뢰도 y 가 주어지면 $x \in P$ 에서 yAx 를 최대화하는 x 를 찾는다.
- 이 x 를 통해서 각 전문가의 성공률을 계산한다. i 번 전문가의 성공률은 A_ix-1 이다. **성공률에 반비례하게** 신뢰도를 갱신한다. 정확히는, 신뢰도에 $1/e^{A_ix-1}$ 을 곱한다.
- 구한 모든 x 의 평균을 답으로 반환한다.

우리는 A_ix 를 1 이상으로 만드는 것이 목표이고 그러한 차원에서 A_ix-1 을 *성공률* 이라고 불렀다. 하지만 성공한 전문가들의 신뢰도는 감소한다. 이 부분이 MWU Framework의 묘미이다. 큰 틀에서, 우리는 A_ix 가 1 이상이기를 바라는 것이지 각각이 큰 값을 원하는 것이 아니다. Oracle은 y 가 클 수록 A_ix 를 키워주는 경향을 가진다. 만약 어떠한 전문가가 A_ix 의 값이 낮다면, 이들은 이후 A_ix 값이 높게 부여되도록 Oracle의 조정을 거친다. 모든 작업을 반복하다 보면 A_ix 값은 전반적으로 *공평해지는* 경향성을 띈다. iteration의 역사를 보면 A_ix 가 작은 적도 있지만, 작을 경우 커지고 클 경우 작아질 것이다. 고로 이들의 평균을 취하면, LP가 Solvable하다는 가정 하에 두 A_ix 가 적당히 1 이상의 값을 가진다.

이제 MWU Framework의 요약이 끝났으니 Garg-Konemann 으로 돌아오면 알고리즘의 몇가지 선택을 이해할 수 있다. 우리는 $Ax \leq b$ 에 대해서 $max(c^Tx)$ 를 해결해야 한다. 일반성을 잃지 않고 $Ax \leq 1$ 이라고 하고, $c^Tx \geq K$ 가 가능한지를 보는 결정 문제로 변환하자. MWU로 돌아와서 Convex space P를 $c^Tx \geq K$ 라고 하자. P 에서 yAx 를 최소화하는 것은,

$\sum(y_iA_{i,j})x_i$ 를 최소화하는 것이다. x_i 를 1 올리면 $\sum y_iA_{i,j}$ 의 비용이 들고 c_i 만큼 목적을 달성한다. 그냥 이 가격 대비 성능비를 최대화하는 행 하나만 고르면 된다. 그리고 그것은 정확히 $length(j)$ 를 최소화하는 Column과 동일하다! 즉, $length(j)$ 를 최소화하는 Column을 고르는 것은 Oracle을 호출하는 것과 동일하다.

이후 알고리즘의 Primal 값은 실제 MWU의 Primal에 대응되고, Dual 값은 MWU의 y 벡터 (*전문가의 신뢰도*) 에 대응된다. 수치에 차이는 있지만, Garg-Konemann x_q 를 특정 수만큼 올리고, 신뢰도 역시 특정한 수치만큼 곱해주는 방법을 택함으로써 MWU와 비슷한 접근을 취한다. 사실, Garg-Konemann의 선택이 일반적인 MWU보다 더 효율적이다 (width-independent).

Proof

이하의 내용은 1번에서 Multi-Commodity Flow를 사용했을 때의 증명과 **동일하다.** 고로 짧게만 설명한다. 매 k 번째 iteration에서

$D(k) = \sum_i b_iy_k(i) = \sum_i b(i)y_{k-1}(i) + \epsilon\frac{b(p)}{A(p,q)}\sum_i A_(i,q)y_{k-1}(i) = D(k-1) + \epsilon(f_k - f_{k-1})\alpha(k-1)$

고로 $D(k) = D(0) + \epsilon\sum_{l=1}^{k}(f_l - f_{l-1})\alpha(l-1)$ 이다.

$\beta = \min_y D(y)/\alpha(y)$ 라고 하자. $\beta \leq D(l-1)/\alpha(l-1)$ 이므로

$D(k) = m\delta + \frac{\epsilon}{\beta}\sum_{l=1}^{k}(f_l - f_{l-1})D(l-1)$

고로 $D(k) \leq m\delta e^{\epsilon f_k}\beta$ 이고, 종료 조건에 의해 $D(t) \geq 1$ 이니

$\frac{\beta}{f_t} \leq \frac{\epsilon}{ln(m\delta)^{-1}}$

이다. 이제 Main Claim을 소개하는데, 증명이 동일하기 때문에 따로 짚지는 않는다.

Claim. $\frac{f_t}{\log_{1+\epsilon}\frac{1+\epsilon}{\delta}}$ 의 값을 가지는 Packing LP의 해가 존재한다.

$\delta = (1+\epsilon)((1+\epsilon)m)^{-\frac{1}{\epsilon}}$ 으로 대입할 경우, 여기서도 Primal과 Dual의 비율이 $\leq (1-\epsilon)^{-2}$ 이 되고, 고로 오차율도 그러하다.

3. Kent's Algorithm for Mixed Packing/Covering Problem

Kent's Algorithm은 Packing/Covering LP를 해결한다. Packing/Covering은 편의상 Optimization의 관점에서 보지는 않고 Decision problem의 관점에서 볼 것이다. (답에 대한 이분 탐색을 통해서 Optimization problem으로 전환할 수 있다.) 고로 Dual도 따로 정의하지는 않으나, 알고리즘 자체는 Primal/Dual 값을 모두 관리하기는 한다.

Primal. Find x such that $Ax \leq b, Cx \geq d$

일반성을 잃지 않고, 여기서는 $b = d = 1$ 이라고 가정한다 (A, C 를 그만큼 scale down하면 된다.)

Algorithm

Partition function

$\pi = \mathbb{R}^n \to \mathbb{R}$ 에 대해 **partition function** $\pi(x) = \log(\sum e^{x_i})$ 를 정의하자.

다음과 같은 사실들을 쉽게 알 수 있다:

- $\max x_i \le \pi(x) \le \max x_i + \log n$
- $\eta = \log n / \epsilon$ 에 대해 $\max x_i \le \frac{1}{\eta}\pi(\eta x) \le \max x_i + \epsilon$
- $\frac{d\pi}{dx_i} = \frac{e^{x_i}}{\sum_j e^{x_j}}$
- $\pi'(x) \times 1 = 1$

Continuous greedy algorithm

어떠한 중간 결과 x 가 있다고 했을 때, 우리가 관심있는 것은 *tightest packing constraint* $\max_i (Ax)_i$ 와 *most deficient covering constraint* $\min_i (Cx)_i$ 이다. Partition function의 성질을 활용하여 이를 ϵ-addictive approximate 한 선에서 다시 써 줄 수 있는데,

$f_p(x) = \frac{1}{\eta}\pi(\eta Ax), f_c(x) = -\frac{1}{\eta}\pi(-\eta Cx)$

라고 써 주면 각각이 $\max_i (Ax)_i$ 와 $\min_i (Cx)_i$ 에 대응된다.

결국 원하는 end condition은 $\max_i (Ax)_i \le \min_i (Cx)_i$ 와 동치라고 볼 수 있다. 저 사이에 숫자 $\max_i (Ax)_i \le K \le \min_i (Cx)_i$ 가 존재하면 $\frac{x}{K}$ 를 답으로 가져가면 그만이기 때문이다. 달리 말해 지금 x 가 해가 아니라는 것은 $max > min$ 임을 뜻한다. 고로 x 를 적당히 조정해서 max 보다 min 이 더 빠르게 증가하도록 하는 것이 우리의 목표이다.

여기서 max 항을 *partition function* 으로 대체한 이유를 알 수 있는데, *partition function*은 미분이 가능하다. 고로 어떤 component i 를 변화시켰을 때 정확히 어느 정도 증가하고 감소하는 지를 근사적으로 추정할 수 있다. 미분을 해 보면

$f_p'(x) = A^T\pi'(\eta Ax), f_c'(x) = C^T\pi'(-\eta Cx)$

이 우리는 적당한 *이동 방향* dx/dt 를 찾아서 $\langle \pi'(-\eta Cx), C(dx/dt) \rangle \approx \frac{d}{dt} min_i (Cx)_i$ 가 $\langle \pi'(\eta Ax), A(dx/dt) \rangle \approx \frac{d}{dt} max_i (Ax)_i$ 보다 크게 하는 것이 목표이다. $y = dx/dt$ 라 하면, 우리가 찾는 것은

$\langle \pi'(-\eta Cx), Cy \rangle \ge \langle \pi'(\eta Ax), Ay \rangle$

를 만족하는 y 이다. 만약 위 식이 만족하고 두 값 사이에 k 라는 실수가 존재한다면, $y := \frac{y}{k}$ 를 대입하여

$\langle \pi'(-\eta Cx), Cy \rangle \ge 1$

$\langle \pi'(\eta Ax), Ay \rangle \le 1$

을 만족하는 y 를 찾는 문제로 생각하여도 무방하다. x 가 주어졌을 때, 이러한 y 를 찾는 문제를 **Oracle** 이라고 하자. (이 Oracle을 구하는 방법은 이후 서술한다.) 만약 Oracle이 주어진다고 하면 다음과 같은, 무한한 시간 복잡도의 알고리즘을 찾을 수 있다.

- 초기 $x = 0$ 으로 둔다.
- $Cx \ge 1$ 이 아닐 때까지 다음을 반복한다:
- Oracle을 통해서 y 를 찾는다.
- y 의 방향으로 무한히 작게 x 를 이동시킨다.
- x 를 반환한다.

이 알고리즘이 y 를 항상 찾을 수 있다면, 이 알고리즘을 통해서 찾은 답은 항상 올바르다.

Lemma 1. 임의의 순간에 $\max_i (Ax)_i \le (1+\epsilon)\min_i (Cx)_i + (2+\epsilon)\epsilon$ 을 만족한다. (증명은 어렵지 않으니 생략)

하지만 해가 있는 인스턴스에서 항상 위 알고리즘이 작동하는지는 불명확하다. 기본적으로 $Ax \leq 1$ 조건이 있기 때문에 정말 "무한히 작게" 이동하는 게 아닌 이상 알고리즘이 항상 종료하는 것은 맞다. 고로 해가 있는데도 알고리즘이 성공하지 않는다면 Oracle이 적절한 y 를 찾지 못한다는 것이다. 그러한 경우에 대해서 따져보기 위해서 Oracle을 구현하는 법을 생각해보자. Oracle은 x 가 주어질 때 다음과 같은 y 를 찾아야 한다.

$\langle \pi'(-\eta Cx), Cy \rangle \geq 1$

$\langle \pi'(\eta Ax), Ay \rangle \leq 1$

Oracle을 구현하는 것은 사실 엄청 쉽다. 이 문제 자체를 일종의 Fractional Knapsack이라고 생각할 수 있는데, y 의 Component y_i 를 1만큼 올리는데 $\langle \pi'(\eta Ax), Ae_i \rangle$ 의 비용이 들고, 그를 통해서 $\langle \pi'(-\eta Cx), Ce_i \rangle$ 의 이득을 얻을 수 있다. 이 때 1의 비용으로 최대한의 이득을 얻어야 하는데, fractional solution이 허용되기 때문에 단순히 *가격 대비 성능비* 가 가장 높은 단 하나의 컴포넌트에 몰아주면 된다. 즉

$\frac{\langle \pi'(-\eta Cx), Ce_i \rangle}{\langle \pi'(\eta Ax), Ae_i \rangle}$

이라는 값 (이를 ***bang-for-buck ratio*** 라고 하자) 이 최대인 i를 찾아야 하고, 그 때의 값이 1 이상이어야 한다. 이제 다음 사실이 성립한다 (증명 생략).

Lemma 2. 만약 전체 문제가 Feasible하다면 임의의 x 에 대해서 $\langle \pi'(\eta Ax), Ae_i \rangle \leq \langle \pi'(-\eta Cx), Ce_i \rangle$ 인 i 가 존재한다.

이를 통해서 Continuous greedy algorithm의 완벽한 명세를 얻었으며 이것이 항상 올바른 답을 찾는다는 증명도 얻을 수 있다.

Kent's algorithm in MWU Framework

지금까지 Kent's algorithm을 MWU Framework와 독립적인 방법으로 유도했지만, potential function의 모양새와 Oracle의 존재로 유추할 수 있듯이, Kent's algorithm 역시 MWU Framework 안에= 속하는 알고리즘이다. MWU 알고리즘에서는 *전문가의 신뢰도* 로 대응되는 Dual variable y 를 관리하며, y 의 각 항은 e^{x_i} 의 꼴을 한다. 즉 partition function의 도함수가 이 dual variable y 에 대응되는 것이다.

예를 들어, $v_i(x) = e^{\eta(Ax)_i}$ 라고 하면, potential function $\pi'(\eta Ax)_i = \frac{v_i(x)}{\langle v(x), 1 \rangle}$ 이라고 쓸 수 있다. 비슷하게, $w_i(x) = e^{\eta(Cx)_i}$ 라고 하면, $\frac{w_i(x)}{\langle w(x), 1 \rangle}$ 이라고 쓸 수 있다.

이 관점으로 보면, v, w 라는 함수로 Dual variable을 조금 더 직접적으로 관리할 수 있고 notation도 훨씬 더 간단해진다. 예를 들어, Oracle 에서 찾는 *가격 대비 성능비* 가 최대인 컴포넌트는 다음을 최대화하는 j 랑 똑같다:

$\frac{\langle w(x), Ce_j \rangle}{\langle v(x), Ae_j \rangle}$

그리고 해가 존재할 조건, 즉 bang-for-buck ratio가 1 이상일 조건은:

$(1+\epsilon)\frac{\langle w(x), Ce_j \rangle}{\langle v(x), Ae_j \rangle} \geq \frac{\langle w(x), 1 \rangle}{\langle v(x), 1 \rangle}$

만약에 매 순간 $v(x), w(x)$ 를 가지고 있다면 이러한 coordinate를 찾는 것은 Matrix 크기에 선형인 시간에 가능하다. 특히, 만약 matrix가 sparse하다면, 0이 아닌 값들의 개수에 비례하게 문제를 해결할 수 있다. $v(x), w(x)$ 를 관리하는 것이 문제인데, 이는 구한 y를 토대로 x 를 이동시킬 때 $v_i(x)$ 에 $e^{\eta(Ay)_i}$ 를 곱하는 (그리고 $w_i(x)$ 에도 비슷한 작업을 하는) 식으로 쉽게 관리할 수 있다. Dual variable에 exponential한 항을 곱해서 관리한다는 점에서, MWU algorithm의 일반적인 흐름을 그대로 따라간다고 볼 수 있다.