VOD 시스템에서 CPU 가용성을 최대화하는 저장공간관리 알고리즘

(An Algorithm for Managing Storage Space to Maximize the CPU Availability in VOD Systems)

정 지 찬 † 고 재 두 † 송 민 석 †† 심 정 섭 ††
(Jichan Jung) (Jaedoo Go) (Minseok Song) (Jeong Seop Sim)

요 약 VOD 서버에 서비스를 요청하는 단말장치의 종류가 다양해짐으로 인해 VOD 서비스 사업자가 제공해야 하는 해상도 버전의 종류 역시 다양해지고 있다. 단말장치가 서비스를 요청할 때 서버는 단말장치에 맞는 해상도 버전으로 서비스를 제공해야 하는데 대개의 경우 서버의 저장공간의 용량이 제한되어 있기 때문에 비디오별로 모든 해상도 버전들을 저장하고 있기는 어렵다. 단말장치가 서버에 저장되어 있는 해상도 버전을 요청한 경우라면 바로 서비스가 가능하다. 하지만 단말장치가 서버에 저장되어 있지 않은 해상도 버전을 요청했다면 저장되어 있던 버전을 이용해 해상도를 변환한 후 서비스를 해주어야 한다. 만약 서버가 해상도를 변환하는 빈도가 높아 CPU 가용성이 충분하지 않다면 다른 단말장치들의 서비스 요청에 바로 응할 수 없게 된다. 따라서 서버에 저장되는 파일들을 CPU 사용률을 줄일 수 있는 해상도의 버전들로 저장하여 CPU 가용성을 높인다면 보다 많은 단말장치의 요청을 허용할 수 있을 것이다.

본 논문에서는 한정된 저장용량을 가진 VOD 서버가 단말장치의 서비스 요청들을 가능한 많이 허용하기 위해 저장해야 할 각 비디오의 해상도 버전들을 분기한정 기법을 이용하여 찾는 알고리즘을 제시한다.

키워드 : 트랜스코딩, VOD, 분기한정 기법, CPU 가용성, 저장공간

Abstract Recent advances in communication and multimedia technologies make it possible to provide video-on-demand(VOD) services and people can access video servers over the Internet at any time using their electronic devices, such as PDA, mobile phone and digital TV. Each device has different processing capabilities, energy budgets, display sizes and network connectivities. To support such diverse devices, multiple versions of videos are needed to meet users' requests. In general cases, VOD servers cannot store all the versions of videos due to the storage limitation.

When a device requests a stored version, the server can send the appropriate version immediately, but when the requested version is not stored, the server first converts some stored version to the requested version, and then sends it to the client. We call this conversion process transcoding. If transcoding occurs frequently in a VOD server, the CPU resource of the server becomes insufficient to response to clients. Thus, to admit as many requests as possible, we need to maximize the CPU availability.

In this paper, we propose a new algorithm to select versions from those stored on disk using a branch and bound technique to maximize the CPU availability. We also explore the impact of these storage management policies on streaming to heterogeneous users.

Key words : transcoding, VOD, branch and bound, CPU availability, storage space

· 이 논문은 인하대학교의 지원에 의하여 연구되었음

† 학생회원 : 인하대학교 컴퓨터정보공학부
jchjung@inhaian.net
goldedit@gmail.com
†† 종신회원 : 인하대학교 컴퓨터정보공학부 교수
mssong@inha.ac.kr
jssim@inha.ac.kr

논문접수 : 2008년 8월 25일
심사완료 : 2009년 1월 31일

정보과학회논문지: 시스템 및 이론 제36권 제3호(2009.6)

1. 서 론

VOD(Video-on-demand) 서버에 서비스를 요청하는 단말장치의 종류가 다양해짐으로 인해 VOD 서비스 사업자가 제공해야 하는 비디오들의 해상도의 종류 역시 다양해지고 있다. 만약 이종의(heterogeneous) 단말장치들이 자신에게 적합한 해상도 버전을 요청했을 때, 서버가 이를 제공할 수 없다면 VOD 서비스 이용에 제약이 발생한다. 예를 들어, 단말장치의 다양화 측면에서 노트북, PDA, PMP 등의 단말장치가 서비스를 요청했을 때 서버는 이들이 요청한 해상도에 맞는 비디오를 제공할 수 있어야 한다. 또한 네트워크의 대역폭(bandwidth) 측면에서 볼 때 단말장치가 데스크톱 정도의 고성능이고 네트워크의 대역폭이 고화질을 즐기기에 충분하다면 굳이 저화질의 해상도를 요청하지는 않을 것이다. 따라서 서버는 이처럼 다양한 경우에 적합한 해상도 버전을 제공해 주어야 한다[1-4].

VOD 서버가 서비스를 하기 위해서 각 비디오 별 모든 해상도 버전을 저장하고 있다면 어떠한 요청에도 바로 서비스를 해줄 수 있지만, 일반적으로 저장공간의 한계로 서버에 모든 해상도 버전을 저장하고 있기는 어렵다. 따라서 일부 해상도 버전만을 저장한 후 서비스를 한다. 만약 요청 받은 해상도가 저장되어 있는 버전이라면 바로 서비스를 해주면 되지만, 그렇지 않은 경우에는 저장되어 있는 버전을 요청받은 해상도 버전으로 변환하여 서비스해야 한다(그림 1 참조). 일반적으로 요청받은 버전보다 가장 비슷한 상위(고화질) 버전을 이용하여 변환하는 것이 CPU 사용률을 줄일 수 있다. 이런 변환 작업을 트랜스코딩(transcoding)이라고 한다[1,2,5,6]. 트랜스코딩은 CPU 사용률이 높은 작업이기 때문에 각 비디오에 대한 선호도 및 트랜스코딩이 일어나는 빈도 등을 고려하여 해상도 버전을 선택하여 저장하는 것이 좋다.

이종의 단말장치들에게 멀티미디어 파일을 지원하기 위해 여러 기법들이 연구되어 왔다[7]. 대표적으로 하나의 비디오 스트림으로 다양한 전송 네트워크와 다양한 수신 단말장치에 적응적 서비스가 가능하도록 하는 Scalabe Video Coding(SVC)이라는 비디오 부호화 기법과, 여러 개의 스트림을 이용하여 서비스하는 트랜스코딩 기반의 기법 등이 있다. SVC 기법은 트랜스코딩을 하는 방식에 비해 저장공간의 사용량이 적다. 하지만 압축률이 효율적이지 못하고 현재의 대부분의 멀티미디어 파일이 non-scalable 형식을 따르고 있기 때문에 트랜스코딩 기법을 이용하는 것이 보다 실용적인 것으로 알려져 있다[2]. [5]에서는 멀티미디어 파일마다 일부의 해상도 버전만을 저장한 후, 저장되어 있는 버전이 요청된 경우 바로 서비스를 하고 그렇지 않은 경우 저장되어 있는 파일을 트랜스코딩하여 서비스하는 기법이 제시되었다. 이 기법에서는 시스템 자원을 탄력적으로 사용할 수 있지만, CPU 가용성(availability)과 저장공간(hard disk)을 최적화하여 사용하지는 못했다. [8]에서는 [5]와 같은 방식의 트랜스코딩을 지원하는 서버에서 각 비디오 별로 CPU 가용성을 최대화하는 해상도 버전집합을 찾는 문제인 최적버전집합선택 문제를 제시하였고 이를 동적프로그래밍 기법을 이용하여 해결하는 알고리즘을 제시하였다.

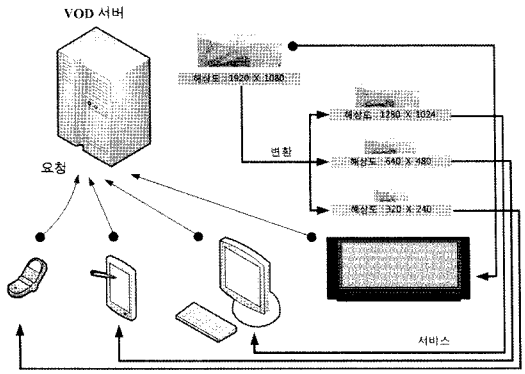


그림 1 트랜스코딩을 지원하는 서버

본 논문에서는 [8]에서 제시한 최적버전집합선택 문제를 분기한정 기법 기반으로 해결하는 새로운 알고리즘을 제시하고, 다양한 실험을 통해 [8]의 알고리즘과 비교 분석한 결과를 제시한다. [8]에서 제시된 알고리즘은 동적프로그래밍 기법의 특성상 비디오의 수가 증가함에 따라 계산에 필요한 주기억장치(main memory)의 양이 증가하기 때문에 주기억장치의 크기에 대한 제약이 크다. 본 논문에서는 우선순위큐와 최선우선탐색을 이용하여 비디오의 수가 일정 크기 이상 많아졌을 때도 해결할 수 있음을 보인다.

본 논문의 구성은 다음과 같다. 2장에서는 관련연구 및 용어들을 설명하고 3장에서는 알고리즘을 제시한다. 4장에서는 제시한 알고리즘을 구현하여 실험한 결과를 보이고 5장에서 결론을 제시한다.

2. 관련 연구

2.1 최적버전집합선택 문제

본 논문에서 다룰 문제를 정의하기 위해 먼저 VOD 시스템을 모델링한다. 여기에서 사용하는 시스템 모델은 [8]에서 정의된 내용을 따른다. VOD 서버에 서비스를 위해 저장해야 할 비디오 파일의 수를 NV라고 하고 각각의 비디오를 $V_i(i = 1, ..., NV)$라고 하자. 각 비디오 별 저장할 수 있는 최대 해상도 버전의 수를 NR이라고 할 때 해상도의 종류에 따른 각 비디오 파일을 V_i^k

표 1 NV = 3이고 NR = 3인 경우 저장공간의 총용량을 넘지 않도록 저장한 예

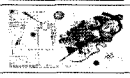
	original (Full HDTV)	PDA	핸드폰
	V_i^1	V_i^2	V_i^3
V_1			
V_2			
V_3			

표 2 NR = 3일 때, FS_1의 원소들

	V_1^1	V_1^2	V_1^3
$FS_{1,1}$			
$FS_{1,2}$			
$FS_{1,3}$			
$FS_{1,4}$			

(k = 1, ..., NR)로 나타낼 수 있다. VOD 서버의 모든 비디오는 표 1과 같이 원본(original version) V_i^1을 반드시 가진다. V_i^1은 최고 해상도 버전이며 다른 해상도에 비해 가장 많은 저장공간을 필요로 한다. V_i^2부터 V_i^{NR}까지는 해상도 및 저장공간의 크기가 점차적으로 줄어든다.

서버의 저장공간의 크기를 넘지 않도록 하는 각 비디오 별 해상도의 조합은 다양할 수 있다(표 1 참조). V_i에서 NR개의 해상도 버전을 조합하여 만들 수 있는 집합들의 모임을 FS_i라 하자. FS_i의 원소의 수를 NE로 나타낼 때, 원본은 반드시 FS_i의 어떠한 원소에도 포함되어야 하므로 $NE = 2^{NR-1}$이 된다. FS_i의 각 원소를 가리킬 때 각각을 버전집합 $FS_{i,j}$(j = 1, 2,..., NE)라 하자. $FS_{i,j}$는 각각의 저장공간의 크기에 따라 오름차순으로 정렬이 된다고 가정한다. 예를 들어, 표 2는 표 1의 첫 번째 비디오의 모든 버전집합 $FS_{1,j}$(j=1,...,NE)를 저장공간의 크기에 따라 오름차순으로 도표화한 것이고 FS_1을 다음과 같이 나타낼 수 있다. FS_1 = {$FS_{1,1}$(={ V_1^1}), $FS_{1,2}$(={ V_1^1, V_1^3}), $FS_{1,3}$(={ V_1^1, V_1^2}), $FS_{1,4}$(={ V_1^1, V_1^2, V_1^3})}.

$S_{i,j}^k$는 버전집합 $FS_{i,j}$에서 비디오 파일 V_i^k의 포함 유무를 나타낸다. 즉, V_i^k가 $FS_{i,j}$에 존재하면 S_{ij}^k = 1, V_i^k가 $FS_{i,j}$에 존재하지 않으면 $S_{i,j}^k$ = 0이다. $C_{i,j}^k$는 $FS_{i,j}$에서 어떤 비디오 파일을 이용하여 V_i^k를 트랜스코딩하여 생성하는데 필요한 CPU 사용률을 나타낸다. $S_{i,j}^k$ = 1이면 $C_{i,j}^k$ = 0이고 바로 서비스를 제공할 수 있다. 그 이외의 경우에는 CPU 사용률을 가장 적게 할 수 있는 버전 즉, 가장 비슷한 상위 버전을 선택하여 트랜스코딩 후 서비스를 제공해야 한다. 각 비디오의 모든 V_i^k에 대한 접속 확률(access probability)이 미리 알려져 있다고 하고 V_i^k에 대한 접속 확률이 P_i^k일 때, $\sum_{i=1}^{NV}\sum_{k=1}^{NR} P_i^k = 1$이다. $FS_{i,j}$를 저장했을 때의 평균 CPU 사용률을 나타내는 $CU_{i,j}$는 $CU_{i,j} = \sum_{k=1}^{NR}(P_{i,j}^k \times C_{i,j}^k)$이 된다.

$SV_{i,j}$는 저장공간에 $FS_{i,1}$을 저장했을 때 보다 $FS_{i,j}$를 대신 저장했을 때, 절약되는 CPU 사용률인 CPU 가용성을 말한다. $FS_{i,1}$을 저장한다는 것은 원본버전인 V_i^1만을 저장하고 있다는 것이고 이는 V_i^1을 제외한 다른 모든 해상도 버전은 V_i^1에서 트랜스코딩을 통해 생성하여 서비스를 제공해 주어야 함을 의미한다. 반면에 $FS_{i,j}$를 저장할 경우 원본 이외에 다른 해상도의 비디오 파일을 저장하고 있기 때문에 $FS_{i,1}$을 저장하고 있을 때보다는 트랜스코딩이 발생하지 않음으로써 CPU 사용률이 절약됨을 알 수 있다. 따라서 $SV_{i,j} = CU_{i,1} - CU_{i,j}$이고 $SV_{i,1} = 0$이다.

서버의 저장공간의 총용량을 TS라고 하고 $FS_{i,j}$를 저장하는데 필요한 용량을 $STR_{i,j}$라고 하자. 총용량 TS를 넘지 않는다는 조건하에서 각 비디오 별로 CPU 사용률을 최소화하는 버전집합 $FS_{i,j}$을 찾아 $FS_{i,j}$의 V_i^k들로 저장해야 한다. 앞으로 이 $FS_{i,j}$의 j를 SE_i(SE_i = 1,..., NE)로 나타낸다. 각각의 FS_i에서 SE_i를 찾는 문제를 최적버전집합선택 문제라 하고 다음과 같이 정의한다.

정의 1. 최적버전집합선택 문제

모든 FS_i($1 \le i \le NV$)가 주어졌을 때, $\sum_{i=1}^{NV} STR_{i,SE_i} \le TS$를 만족하며 CPU 가용성($\sum_{i=1}^{NV} SV_{i,SE_i}$)을 최대가 되도록 하는 SE_i(SE_i = 1,..., NE)를 찾는 문제를 최적버전집합선택 문제라 한다.

각각에 대해 이익(profit)과 무게(weight)가 주어지는 n개의 아이템과 용량 c인 배낭이 주어졌을 때, 아이템들의 무게의 합이 배낭의 용량을 초과하지 않으면서 그 이익의 합이 최대가 되도록 하는 문제를 0-1 배낭문제(0-1 knapsack problem)라 한다[9]. 0-1 배낭문제는 잘 알려진 NP-hard 문제이다[9]. 한편, 이를 확장하여 각각이 이익($p_{i,j}$)과 무게($w_{i,j}$)를 가지는 아이템(n_j)들로 구성된 서로 다른 m개의 클래스($N_1,\ldots,N_m$)와 용량이 c인 배낭이 주어졌을 때, 각 클래스에서 정확히 하나씩의 아이템을 선택하여 아이템들의 무게의 합이 배낭의 용량을 초과하지 않으면서 그 이익의 합이 최대가 되도록 하는 문제를 다중선택배낭 문제(Multiple Choice Knapsack Problem)라 하는데 이 문제 역시 잘 알려진 NP-hard 문제이다[9](그림 2 참조).

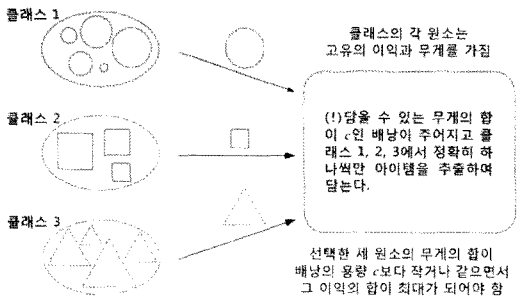


그림 2 세 개의 클래스가 주어진 다중선택배낭 문제

본 논문에서 다룰 최적버전집합선택 문제는 서버의 저장공간의 총용량을 나타내는 TS, 각 비디오별 선택 가능한 버전집합들이 담긴 FS_i, 각 버전집합에서 CPU 가용성인 $SV_{i,j}$, 각 버전집합에서의 용량인 $STR_{i,j}$가 각각 배낭의 용량 c, 클래스 N_i, 이익 $p_{i,j}$, 무게 $w_{i,j}$에 해당하는 다중선택배낭 문제이므로 역시 NP-hard 문제이다.

2.2 분기한정 기법

최적화 문제의 일종인 다중선택배낭 문제의 최적해(solution)는 상태공간트리를 이용하여 찾을 수 있다. 상태공간트리는 최적화 문제의 해가 될 수 있는 후보들의 모임 즉, 해공간을 나타내는 트리이다. 상태공간트리에서 루트(root)부터 리프(leaf)까지의 경로 상에 있는 노드들을 선택하게 되면 하나의 해답후보가 된다. 예를 들어, 그림 3은 NV = 3, NR = 3일 때의 상태공간트리로써 최적해를 찾기 위해 모든 해답후보를 탐색하는 과정이며, 만약 $N_{3,2}$를 선택하였을 때 루트에서부터 $N_{3,2}$까지의 경로 상에 있는 모든 노드들 $N_{1,1}$, $N_{2,1}$, $N_{3,2}$은 하나의 해답후보가 된다.

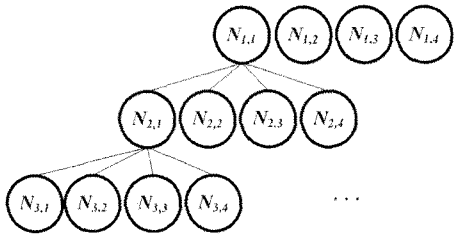


그림 3 모든 해답후보를 생성하여 최적해를 찾는 과정

상태공간트리에서 루트부터 모든 리프까지의 경로를 확인하면 최적해를 찾을 수 있다. 따라서 다중선택배낭 문제의 최적해를 찾기 위해 확인해야 할 해답후보의 수는 $\prod_{i=1}^{m}|N_i|(=|N_1|\times\cdots\times|N_m|)$이 된다. 이 해답후보를 줄이기 위해 다음의 단순 우성의 개념을 이용한다.

정의 2. 단순 우성(simple dominance)

동일한 클래스 N_i 안에 존재하는 두 개의 아이템 j와 k가 다음을 만족하면 아이템 j는 아이템 k에 대해 단순 우성이라고 한다.

$$w_{i,j} \le w_{i,k} \text{ 이고 } p_{i,j} \ge p_{i,k}.$$

단순 우성의 개념은 두 개의 아이템에 대해 더 무거운 아이템이 더 적은 이익을 가질 때 최적해를 찾는데 우성인 아이템만을 이용하자는 것이다. 이는 우성인 아이템을 포함하는 최적해가 존재하기 때문이다[9]. 예를 들어, 그림 4는 같은 클래스 내의 8개의 아이템을 무게-이익 좌표평면에 나타낸 것이다. 그림 4에서 4번 아이템은 2번 아이템보다 무겁지만 이익은 더 적다. 즉, 4번 대신 2번을 선택하면 더 적은 무게로 높은 이익을 얻을 수 있으므로 최적해를 얻기 위해서 4번 아이템은 선택할 필요가 없다. 그림 4에서 6번, 8번 아이템 역시 마찬가지로 2번 아이템보다 무겁지만 이익은 더 적으므로 선택할 필요가 없다. 그림 4에서 검은 색으로 표시된 1번, 2번, 3번, 5번, 7번 아이템이 다른 아이템들에 대해

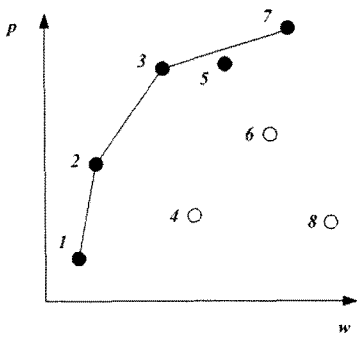


그림 4 단순 우성 아이템들(검은 색으로 표시)

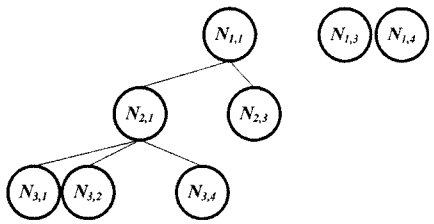


그림 5 단순우성 아이템만을 이용하는 상태공간트리

단순 우성이므로 나머지 4번, 6번, 8번 아이템은 상태공간트리에서 삭제한다.

단순우성의 개념을 그림 3의 세 개의 클래스에 적용하여 단순우성이 아닌 아이템들을 삭제하면 그림 5와 같이 최적해를 찾기 위해 생성해야 할 해답후보의 수가 줄어들게 된다.

위의 방식으로는 노드의 수가 줄어들었다 하더라도 여전히 큰 계산량이 필요할 수 있기 때문에 최적해를 찾기 위해 분기한정 기법(branch and bound)을 이용할 수 있다[9]. 분기한정 기법을 통한 다중선택배낭 문제를 해결하는 알고리즘을 Dyer, Kayal and Walker가 제시하였다[10]. 분기한정 기법은 최적화 문제를 해결하는 기법 중의 하나로 역추적(backtracking)을 이용하는 기법이며 단순 우성 개념을 이용해 노드의 수가 줄어든 상태공간트리를 최선우선탐색(best first search)을 하여 최적해를 찾는다[11]. 최선우선탐색은 각 노드에서 얻을 수 있는 이익의 상한값(upper bound)과 그 노드를 포함한 해가 보장하는 최대이익인 하한값(lower bound)을 계산한다. 이때 상한값과 하한값의 계산에 LP 우성인 아이템을 이용한다. 이는 LP 우성의 개념을 이용하여 축소된 집합에 대해 상한값을 계산해도 축소되기 이전 집합에 대한 상한값보다 작지 않음이 보장되기 때문이다[9].

정의 3. LP 우성(linear programming dominance)

동일한 클래스 N_i 안에 존재하는 세 개의 아이템 j, k, l이 다음을 만족하면 아이템 j와 l은 아이템 k에 대해 LP 우성이라고 한다.

$$\frac{p_{i,l}-p_{i,k}}{w_{i,l}-w_{i,k}} \geq \frac{p_{i,k}-p_{i,j}}{w_{i,k}-w_{i,j}}.$$

LP 우성인 아이템은 그림 4와 같이 주어진 아이템들을 무게-이익 좌표 평면에 표현했을 때 볼록표면(convex hull)을 형성한다. 그림 4에서 볼록표면 위의 1, 2, 3, 7 아이템이 LP 우성 아이템을 나타낸다.

3. 알고리즘

본 논문에서 제시하는 최적버전집합선택 문제 해결 알고리즘은 크게 두 단계로 구성된다. 먼저 상태공간트리 탐색 과정에서의 계산량을 줄이기 위한 해공간을 축소시킨다. 다음으로 상한값과 하한값을 이용하여 최선우선탐색 전략으로 최적버전을 찾는다.

3.1 해공간 축소

해공간 축소는 본 논문에서 앞서 설명한 단순 우성의 개념을 이용하는 과정이다. 본 논문에서 NV개의 FS_i는 NE개의 버전집합으로 구성되고 각 버전집합 $FS_{i,j}$를 이용하여 상태공간트리의 노드를 생성한다. 이때 상태공간트리에서 생성 가능한 최대 해답후보의 수는 NE^{NV}개가 된다. 이는 모든 해답후보들을 확인하여 최적해를 찾을 경우 대단히 큰 계산량이 필요함을 의미한다. 따라서 해공간 축소를 통해 각 FS_i의 원소수를 즉, 노드의 수를 감소시켜 해답후보를 줄이게 된다. 각 FS_i가 다중선택배낭 문제에서 각각의 클래스에 해당하고 $SV_{i,j}$는 이익, $STR_{i,j}$는 무게에 해당하므로 최적버전집합선택 문제에서 단순 우성의 개념은 다음과 같이 정의할 수 있다.

정의 4. 최적버전집합선택 문제에서의 단순 우성

최적버전집합선택 문제에서 동일한 클래스 FS_i 안에 존재하는 두 개의 버전집합 $FS_{i,j}$와 $FS_{i,k}$가 다음을 만족하면 버전집합 $FS_{i,j}$는 버전집합 $FS_{i,k}$에 대해 단순 우성이라고 한다.

$$STR_{i,j} \leq STR_{i,k} \text{ 이고 } SV_{i,j} \geq SV_{i,k}.$$

단순 우성인 버전집합만 남기는 과정은 다음과 같다. 먼저 각 $FS_i(1 \leq i \leq NV)$별로 모든 $FS_{i,j}(1 \leq j \leq NE)$를 저장공간($STR_{i,j}$)에 대해 오름차순으로 정렬한다. 다음으로 저장공간이 가장 적은 버전집합부터 시작하여 연속된 두 버전집합 중 저장공간이 더 큰 버전집합의 CPU 가용성이 높아지지 않으면 저장공간이 더 큰 버전집합을 FS_i에서 삭제한다. 단순 우성 개념을 통해 원소수가 축소된 FS_i를 R_FS_i라고 하고 R_FS_i의 각 원소인 버전집합들을 $R_FS_{i,j}(1 \leq j \leq |R_FS_i|)$라 하자.

3.2 최적버전집합선택

최선우선탐색을 이용하여 각 비디오 별 최적버전집합을 선택하는 과정은 다음과 같다. 먼저 상태공간트리에서 처음으로 분기를 시키는 노드를 찾는다. 이 후 최선우선탐색을 다음의 두 경우 중 하나에 해당하는 노드를 만날 때까지 진행하여 최적버전집합을 찾는다. 첫 번째 경우는 노드에서의 상한값이 버전집합들을 조합하여 얻을 수 있는 최대 이익과 같은 값을 가진 노드를 만나는 경우이다. 두 번째 경우는 리프이면서 하한값이 지금까지 보장해 줄 수 있는 최대 이익을 나타내는 *Max profit* 과 같은 노드를 만나는 경우이다.

각 노드에서의 상한값과 하한값의 계산에 LP 우성인 버전집합을 이용한다. 최적버전집합선택 문제에서의 LP

우성의 개념은 다음의 정의 5와 같고 LP 우성인 버전집합은 각 R_FS_i에서 앞에서 설명한 대로 R_FS_i의 버전집합들을 저장공간(STR)-CPU 가용성(SV) 좌표평면에 표현했을 때 볼록표면을 구성하는 버전집합들을 찾으면 된다. 이렇게 구해진 버전집합들의 모임을 LP_FS_i $(1 \le i \le NV)$라 하고 각각의 버전집합을 $LP\_FS_{i,j}$ $(1 \le j \le |LP_FS_i|)$라 하자.

정의 5. 최적버전집합선택 문제에서의 LP 우성

최적버전집합선택 문제에서 동일한 클래스 R_FS_i 안에 존재하는 세 개의 버전집합 $R\_FS_{i,j}$, $R\_FS_{i,k}$, $R_FS_{i,l}$이 다음을 만족하면 버전집합 $R_FS_{i,j}$와 $R_FS_{i,l}$은 $R_FS_{i,k}$에 대해 LP 우성이라고 한다.

$$\frac{SV_{i,l}-SV_{i,k}}{STR_{i,l}-STR_{i,k}} \ge \frac{SV_{i,k}-SV_{i,j}}{STR_{i,k}-STR_{i,j}}.$$

각 노드에서의 상한값과 하한값은 다음과 같이 그리디 기법을 이용하여 계산한다. LP_FS_i부터 LP_FS_{NV}에서 모든 첫 번째 버전집합 $LP_FS_{i,1}$을 포함시킨 후, 각 LP_FS_i에 남아 있는 버전집합들에서 각 버전집합과 바로 이전 버전집합과의 SV/STR의 비율을 구한다. 모든 LP_FS_i에서 구해진 이전 버전집합과의 SV/STR 비율을 모두 이용하여 내림차순으로 정렬한 뒤 이 값이 가장 큰 버전집합을 선택해 같은 비디오 번호를 가진 기존 버전집합과 바꾸어 나간다. 0-1 배낭문제와 달리 아이템을 분할하여 일부만 선택할 수 있는 문제를 일반 배낭문제라 하는데, 이 문제의 최적해는 그리디 기법을 이용하여 다항식 시간에 찾을 수 있다. 0-1 배낭문제의 최적이익은 일반 배낭문제의 최적이익보다 클 수 없으므로 일반 배낭문제의 최적이익이 0-1 배낭문제의 최적이익에 대한 상한값이 될 수 있다. 그리디 기법에 의해 버전집합을 선택해 나갈 때 버전집합이 완전히 포함될 수는 없어 일부만 포함되어야 할 경우가 있다. 이렇듯 분할되어야 하는 버전집합을 $LP_FS_{a,b}$라 하자. LP_FS_i부터 LP_FS_{NV}까지의 버전집합들 중 그리디 기법을 이용하여 $LP_FS_{a,b}$를 포함시키지 않고 얻은 이익에 루트부터 노드 $R_FS_{i,j}$까지 선택된 버전집합의 이익의 합을 나타내는 $a_SV_{i,j}$를 더하여 하한값으로 한다. 마찬가지로 a_SVi,j에 $LP_FS_{a,b}$를 분할시켜 계산한 최대가능 이익(실수값 일 수 있음)을 더하여 상한값으로 한다. 앞으로 $R_FS_{i,j}$에서 그리디 기법으로 아이템을 쪼개지 않고 이익을 계산하는 것을 $u_bound(i)$라 하고 아이템을 쪼개어 이익을 계산하는 것을 $l_bound(i)$라 하자.

인기도 순으로 내림차순으로 정렬된 NV개의 비디오에서 최적버전집합선택을 하는 알고리즘의 의사코드는 다음과 같다.

```
알고리즘 최적버전집합선택(Version Set Selection Algorithm)
입력: R_FS_{i,j}(1≤i≤NV, 1≤j≤|R_FS_i|)
출력: s_Array[]
1:  Temporary variables: i, j, pq, lower_bound,
    N_frac, idx, sArr_size, s_Array[]
2:  Max_profit ← l_bound(0)
3:  N_frac ← ⌊ u_bound(0) ⌋
4:  idx ← 0, sArr_size ← 1
5:  for j ← 1 to |R_FS_1| do
6:      if a_SV_{1,j} + l_bound(1) > Max_profit
7:      then Max_profit ← a_SV_{1,j} + l_bound(1)
8:      if a_SV_{1,j} + u_bound(1) ≥ Max_profit
9:      then pq.push(R_FS_{1,j}, sArr_size)
10:         (idx, i, j) into s_Array[sArr_size]
11:         sArr_size++
12: end for
13: while(!pq.empty())
14:     idx ← select(pq.top())
15:     pq.pop();
16:     for j ← 1 to |R_FS_{i+1}| do
17:         if a_SV_{i,j} + l_bound(i+1) > Max_profit
18:         then Max_profit ← a_SV_{i+1,j} + u_bound(i+1)
19:         if a_SV_{i+1,j} + u_bound(i+1) ≥ Max_profit
20:         then pq.push(R_FS_{i+1,j}, sArr_size)
21:              (idx, i+1, j) into s_Array[sArr_size]
22:              sArr_size++
23:         if Max_profit = N_frac or
24:         (Max_profit = a_SV_{i+1,j} + u_bound(i+1,j)
            and i+1 = NV)
25:         then return s_Array[]
26:     end for
27: end while
```

1행에서 pq는 각 노드의 상한값을 기준으로 생성된 우선순위큐를 나타낸다. $s_Array[]$는 우선순위큐에서 추출된 노드의 비디오 번호와 버전집합 번호 및 해당 노드를 분기 시킨 노드를 알기 위한 배열이다. 2행에서는 그리디 기법으로 $Max\ profit$을 결정한다. 3행에서는 그리디 기법을 이용하여 버전집합들을 조합하여 얻을 수 있는 최대 이익을 계산하여 N_frac에 저장한다.

최선우선탐색은 가장 큰 상한값을 가지는 노드를 선택하여 탐색을 진행한다. 먼저 첫 번째 비디오의 버전집합들을 탐색하여 처음으로 분기시킬 노드를 찾는다. 각 노드를 탐색할 때 다음사항들을 확인한다. 탐색한 노드에서의 하한값은 그 노드를 포함한 해가 보장하는 최대 이익을 나타내므로 6~7행에서 하한값이 $Max\ profit$보다 큰 경우 그 값을 바꾸어 준다. 상한값이 $Max\ profit$보다 크거나 같은 노드를 탐색에 이용해야 최적해를 찾을 수 있기 때문에 8~9행에서 이를 확인하여 우선순위큐에 삽입한다. 삽입 시 $s_Array[]$의 증가된 index를 알리는 $sArr_size$를 노드에 저장한다. 이때 10행에서

s_Array[]에 우선순위큐에 삽입되는 노드의 비디오 번호와 버전집합 번호 및 분기시킨 노드의 *s_Array*[] 내에서의 index를 삽입한다.

이 후 14행에서 우선순위큐에 삽입된 노드들 중에 가장 큰 상한값을 가진 노드를 선택하고, 해당 노드의 *s_Array*[]에서의 인덱스를 *idx*에 저장하는 *select*(*pq.top*())라는 과정을 거친다. 선택된 노드는 우선순위큐에서 제거되고 $|R_FS_{i+1}|$만큼 자식 노드를 분기 시킨다. 분기되는 노드에서는 앞선 과정과 비슷하게 다음 사항을 확인한다. 17~18행에서 하한값을 이용하여 *Max profit* 갱신여부를 확인하고, 18~19행에서 상한값을 이용하여 우선순위큐에 삽입 여부를 판단한다. 삽입 시 *s_Array*[]의 증가된 index를 알리는 *sArr_size*를 노드에 저장한다. 이때 21행에서 *s_Array*[]에 우선순위큐에 삽입되는 노드의 비디오 번호와 버전집합 번호 및 분기시킨 노드의 *s_Array*[] 내에서의 index를 삽입한다. 이 과정을 우선순위큐에 노드가 없을 때까지 진행하지만 분기를 하면서 최적해를 가진 노드를 발견하게 되면 *s_Array*[]를 출력하고 종료한다. 최적해를 가지는 노드를 만나는 경우는 두 가지이다. 첫 번째는 *Max profit*과 *N_frac*이 같은 노드를 만난 경우이다. 두 번째는 리프이면서 하한값과 *Max profit*이 같은 노드이다. 최적해를 가진 노드를 찾을 때까지 *s_Array*[]의 크기는 지속적으로 증가하는데 만약 미리 설정해둔 한계치까지 커지면 그때까지의 내용을 보조기억장치(hard disk)에 옮겨서 주기억장치(main memory)의 사용량을 줄인다.

최적버전집합의 출력은 종료된 조건에 따라 다음과 같이 나뉜다. 먼저 첫 번째 조건으로 끝난 경우 *NV*번째 비디오에서 최적해를 가진 노드까지는 $u_bound(i+1)$을 계산하는데 이용한 버전집합들을 출력하고, 루트부터 최적해를 가진 노드까지는 *s_Array*[]를 이용하여 출력한다. 두 번째 조건에서는 *s_Array*[]만을 이용하여 각 비디오별 최적버전집합을 출력할 수 있다.

4. 실험 결과 및 분석

본 논문에서 제시된 알고리즘에 대한 실험은 아래의 환경에서 수행되었다.

- 프로세서: Intel Pentium 4(3.0GHz)
- 메인메모리: 2GB
- 운영체제: Linux Fedora Core 7

FFMPEG 프로그램[12]에서 5 가지의 샘플 비디오의 원본을 다른 해상도들로 트랜스코딩 할 때 사용된 CPU 사용률과 변환된 용량을 측정하고 이를 실험의 입력 *SV*와 *STR*로 이용하였다. 실험에는 각각 1,000 개, 1,500 개, 2,000 개의 비디오의 원본을 이용하였다. 이는 5 가지 샘플 비디오를 200번, 300번 400번 반복하여 생성하였고 서로 다른 비디오로 가정한다. 각 비디오별 해상도의 종류(*NR*)는 5로 설정하였다. 실험에 쓰이는 접속자들의 접속 간격은 포아송 분포도(*Poisson distribution*)를 따른다고 가정하고 평균 3초로 하였다. 모든 비디오에 대한 접속 확률은 *Zipf* 분포도(*Zipf distribution*)를 따른다[13].

본 논문에서 제시한 알고리즘의 성능을 평가하고자 최적버전집합선택 알고리즘 및 다음 두 가지 선택 방식에서의 서비스 허용률을 측정하고 비교하였다. 서비스 허용률은 서버가 일정 시간동안 서비스를 요청한 클라이언트들 중에서 정상적으로 서비스를 해준 비율을 말한다. 본 논문에서는 24 시간 동안 28713 명에게 서비스 했을 때의 서비스 허용률을 측정하였다.

- 인기도 우선 선택 방식(PBS): *NV*개의 모든 비디오의 원본들을 저장한 뒤, 비디오별 인기도 순에 따라 저장할 공간이 부족해질 때까지 저장해 나가는 방식이다.
- 무작위 선택 방식(RBS): 인기도 우선 선택과 마찬가지로 *NV*개의 모든 비디오의 원본들을 저장한 뒤 각 비디오들에 대한 버전집합을 무작위로 선택해 저장을 한다. 인기도 우선 선택과 마찬가지로 저장장치의 총용량을 모두 소진 시킬 때까지 이 과정을 반복한다.

실험은 세 가지 경우로 나누어 수행하였다. 첫 번째 실험에서는 비디오의 개수가 1,000 개로 동일하고 모든 비디오에 대한 접속 확률이 *Zipf* 분포도값 α = 0.271을 따를 때 즉, 모든 비디오에 대한 접속 확률이 비디오 인기도가 일반적인 모델을 따르도록 했을 때[14], 저장공간의 총용량 *TS*를 달리하여 각 방식의 서비스 허용률을 측정하였다. 두 번째와 세 번째 실험에서는 각각 저장공간의 총용량으로 0.8 TB(=819,200 MB), 1.2 TB (=1,228,900 MB), 1.6 TB(=1,638,400 MB)가 주어지는 1,000 개, 1,500 개, 2,000 개의 비디오를 이용하여 모든 비디오에 대한 접속 확률이 α = 0.271일 때와 α = 0.95로 하였을 때의 서비스 허용률을 측정하였다. 이때 $\forall i,\ p_i^k = p_i \times \frac{1}{NR} (i = 1, ..., NV)$이다.

그림 6은 첫 번째 실험의 결과값을 나타낸다. 네 가지 용량에서 모두 본 논문에서 제시한 알고리즘이 가장 좋은 서비스 허용률을 보였다. 무작위 선택보다 많게는 11%, 인기도 우선 선택에 비해 4% 정도 허용률이 높게 나왔다. 서비스 허용률은 주어진 용량이 클수록 PBS와의 차이가 줄어들었는데 이는 저장할 용량이 커질수록 트랜스코딩이 적게 일어나기 때문으로 서버의 저장공간이 적을수록 본 논문에서 제시한 알고리즘이 효율적임을 의미한다.

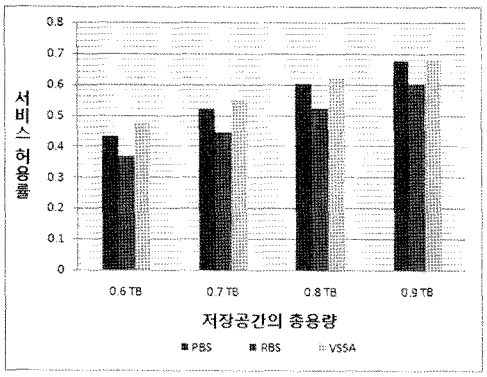


그림 6 *TS*에 따른 서비스 허용률

이 실험에서 최적버전집합을 선택하는데 사용한 주기억장치 사용량을 통해 [8]에서 제시한 동적프로그래밍 기법을 이용한 알고리즘에서의 주기억장치 사용량을 비교할 수 있다. 본 논문의 알고리즘에서 사용한 주기억장치량은 최적해를 찾는 동안 이용된 우선순위 큐에 대한 배열의 크기와 우선순위 큐에서 제거된 노드의 정보를 기억하는 배열의 크기의 더한 것이다. 측정 결과, 서버의 저장공간의 총용량(*TS*)과 비디오의 개수(*NV*)의 곱에 비례하는 크기의 주기억장치를 사용하는 동적프로그래밍 기반의 알고리즘에 비해, 본 논문의 알고리즘이 월등히 적은 양의 주기억장치를 사용함을 알 수 있다(표 3 참조).

그림 7은 두 번째 실험의 결과값을 나타낸 것이고 본 논문에서 제시한 알고리즘의 서비스 허용률이 PBS 보다 많게는 약 2.5% 정도 높음을 알 수 있다. 1,500개와 2,000개의 경우 1,000개에 비해 주기억장치를 아주 적게 사용하였는데(표 4 참조) 이는 최적해를 인기도가 높은 버전집합들을 이용하여 그 값을 구성하였기 때문으로 세 번째 실험을 통해 확인할 수 있다.

표 3 두 가지 방식에서 총용량(*TS*)에 따른 주기억장치 사용량

	0.6 TB	0.7 TB	0.8 TB	0.9 TB
동적프로그래밍	614.4 MB	716.8 MB	819.2 MB	921.6 MB
분기한정	약 30 KB	약 2 KB	약 357 MB	약 2 KB

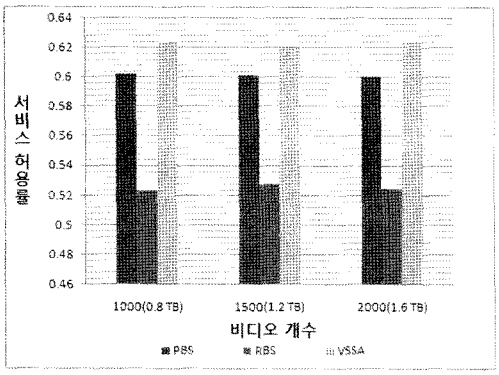


그림 7 *NV*에 따른 서비스 허용률

표 4 비디오 개수(*NV*) 및 *Zipf* 값에 따른 주기억장치 사용량

	1,000개	1,500개	2,000개
α = 0.271	357 MB	약 69 KB,	약 1 KB
α = 0.95	약 1 KB	약 1 KB	약 30 KB

세 번째 실험에서는 *Zipf* 분포도값을 α = 0.95로 하여 접속 확률을 비디오 인기도가 극단적으로 비대칭적인 모델에 비례하도록 하여 즉, 접속확률이 극단적으로 상위 비디오에 집중되도록 하여 인기도의 차이가 확연히 드러나도록 하였다. 세 번째 실험에서는 1,000개의 비디오에서도 메모리 사용량이 아주 적게 사용되었고(표 4 참조) 이를 통해 인기도가 높은 버전집합들을 이용할 경우 최적버전집합을 적은 양의 주기억장치로 훨씬 빨리 찾을 수 있음을 알 수 있다.

비디오의 개수를 달리한 두 번째와 세 번째 실험에서도 [8]에서 제시한 동적프로그래밍 기법을 이용한 기법보다 월등히 적은 양의 주기억장치를 사용함을 알 수 있다. 1,000개, 1,500개에 대해 [8]에서 제시한 알고리즘은 819.2 MB, 1,228.9 MB의 주기억장치를 사용하는데 반해 분기한정 기반일 경우 주기억장치 사용량이 약 1 KB, 약 1 KB 밖에 되지 않았다. 2,000개에 대해서는 주기억장치 사용량을 비교할 수가 없었다. 이는 동적프로그래밍 기법을 이용한 방법이 사용 가능한 주기억장치의 양보다 계산에 이용할 주기억장치의 양이 더 많아서 최적해를 계산할 수 없었기 때문이다. 하지만 본 논문에서 제시된 알고리즘을 이용하면 2,000개에 대해서도 최적해의 계산이 가능하였다. 이는 주기억장치 사용량의 차이에 따른 것으로 동적프로그래밍 기법을 이용할 때보다 많은 비디오에 대해서도 최적해의 계산이 가능함을 나타낸다.

5. 결 론

본 논문에서는 분기한정 기법을 이용하여 동적프로그래밍 기법을 이용한 방법에 비해 훨씬 적은 메모리를 이용하여 최적버전집합선택 문제를 해결하였다. 그러나 동적프로그래밍 기법을 이용한 알고리즘이 일정 시간에 최적해를 찾는데 비해 분기한정 기반 알고리즘의 수행시간은 가변적이고 저장공간의 총용량에 따라 훨씬 많이 소요되는 경우도 있었다.

본 알고리즘을 수행한 실험은 *Zipf* 분포도와 포아송

분포도 등을 따르며 실제 상용 VOD 서비스에 근접한 모델을 이용하여 수행하였다. 실제 상용 서비스의 데이터를 이용한다면 보다 정확한 서비스 허용률을 측정할 수 있을 것이다.

참 고 문 헌

[1] R. Mohan, J. Smith and C. Li, "Adapting multimedia internet content for universal access," IEEE Transactions on Multimedia, Vol.1, No.1, pp. 104-114, March, 1999.

[2] B. Shen, S. Lee and S. Basu, "Caching strategies in transcoding enabled proxy systems for streaming media distribution networks," IEEE Transactions on Multimedia, Vol.6, No.2, pp. 375-386, April, 2004.

[3] X. Tang, F. Zhang, and S. Chanson. Streaming media caching algorithms for transcoding proxies. In Proceedings of the International Conference on Parallel Processing, pp. 287-295, August, 2002.

[4] Tamer Shanableh and Mohammed Ghanbari, "Heterogeneous video transcoding to lower spatio-temporal resolutions and different encoding formats," IEEE TRANSACTIONS ON MULTIMEDIA, Vol.2, No.2, pp. 101-110, June, 2000.

[5] I. Shin and K. Koh, "Hybrid transcoding for QoS adaptive video on demand services," IEEE Transactions on Consumer Electronics, Vol.50, No.2, pp. 732-736, May, 2004.

[6] J. Xin, C. W. Lin and M. T. Sun, Digital video transcoding, pp. 84-97, Proceedings of the IEEE Volume 93, Issue 1, Jan, 2005.

[7] M. Song and H. Shin, "Replication and retrieval strategies for resource effective admission control in multi-resolution video servers," Multimedia Tools and Applications Journal, Vol.28, No.3, pp. 89-114, March, 2006.

[8] M. Song, J.S. Sim, J. Go, B. Lee and S.J. Park, "Balancing MPEG transcoding with storage in multiple-quality video-on-demand services," ETRI Journal, Vol.31, No.3, pp. 333-335, 2009.

[9] Hans Kellerer, Ulrich Pferschy and David Pisinger, Knapsack problems, Springer, 2004.

[10] M. E. Dyer, N. Kayal and J. Walker, "A branch and bound algorithm for solving the multiple choice knapsack problem," Journal of Computational and Applied Mathematics, Vol.11, pp. 231-249, 1984.

[11] Richard Neapolitan and Kumarss Naimipour, Foundations of algorithms. Jones and Bartlett Computer Science, 2004.

[12] http://ffmpeg.mplayerhq.hu/.

[13] A. Dan, D. Sitaram and P. Shahabuddin, "Dynamic batching policies for an on-demand video server," ACM/Springer Multimedia Systems Journal, Vol.4, No.3, pp. 112-121, 1996.

[14] C. C. Aggarwal, J. L. Wolf and P. S. Yu, On optimal batching policies for video on demand storage server, Proceedings of the 1996 International Conference on Multimedia Computing and Systems, p. 253, June, 1996.

정 지 찬

2007년 인하대학교 컴퓨터공학부 컴퓨터공학 학사. 2007년~현재 인하대학교 컴퓨터정보공학 석사과정. 관심분야는 알고리즘, 계산이론

고 재 두

2007년 한국산업기술대학교 컴퓨터공학 학사. 2007년~현재 인하대학교 컴퓨터정보공학 석사과정. 관심분야는 Embedded systems, Embedded software, Low-power storage systems, Real-time-systems and Multimedia systems

송 민 석

1996년 서울대학교 컴퓨터공학과 학사 1998년 서울대학교 컴퓨터공학부 석사 2004년 서울대학교 전기컴퓨터공학부 박사. 2005년~현재 인하대학교 컴퓨터정보공학부 조교수. 관심분야는 실시간 시스템, 임베디드 시스템, 멀티미디어 시스템

심 정 섭

1995년 서울대학교 컴퓨터공학과 학사 1997년 서울대학교 컴퓨터공학과 석사 2002년 서울대학교 전기컴퓨터공학부 박사. 2002년~2004년 한국전자통신연구원(선임연구원). 2004년 9월~현재 인하대학교 컴퓨터정보공학부(조교수). 관심분야는 알고리즘, 최적화이론, 바이오인포매틱스