Uso de *t*-Spanners para Búsqueda en Espacios Métricos ⋆

Rodrigo Paredes

Centro de Investigación de la Web,
Departamento de Ciencias de la Computación,
Universidad de Chile.
Blanco Encalada 2120, Santiago, Chile.
raparede@dcc.uchile.cl

Profesor Guía: Gonzalo Navarro.
gnavarro@dcc.uchile.cl

Resumen

El problema de *Búsqueda en Espacios Métricos* o *Búsqueda Aproximada* consiste en encontrar los elementos de un conjunto más cercanos a una consulta dada bajo algún criterio de similitud. Este problema tiene aplicaciones en muchas áreas de las ciencias de la computación, por ejemplo: bases de datos multimediales, clasificación automática, cuantización y compresión de imágenes, recuperación de texto, biología computacional y predicción de funciones. Por otro lado, en teoría de grafos se tiene el concepto de *t-spanner*, que consiste en un subgrafo G' de un grafo G tal que aproxima las distancias en G con un factor de precisión t.

En este trabajo se propone una nueva metodología para resolver el problema de búsqueda en espacios métricos, la cual considera un t-spanner $G'(V, E)$ como la representación de la base de datos métrica. En G', el conjunto de vértices V corresponde a los objetos del espacio métrico y el conjunto de aristas E corresponde a una selección reducida de las distancias entre pares de objetos.

Para cumplir este objetivo se proponen varios algoritmos de construcción de t-spanners, de inserción y borrado de objetos, de reconstrucción de t-spanners, y de recuperación de objetos. Se evalúan experimentalmente todos los algoritmos propuestos.

Por último, se conjetura que la propiedad del espacio métrico que resulta esencial para el buen desempeño de los t-spanners es la existencia de clusters de elementos, y se entrega suficiente evidencia empírica que avala este resultado. Esta propiedad se presenta en la mayoría de los espacios métricos construidos con objetos del mundo real, por lo que se espera que los t-spanners se comporten bien en la práctica. Además, se muestra que los t-spanners tienen un gran potencial de aplicaciones y mejoras en el ámbito de los espacios métricos.

Abstract

The problem of *Searching in Metric Space* or *Proximity Searching* consists on finding the elements of a set which are closest to a given query under some similarity criterion. This problem has applications in several computer science branches, for example: multimedia databases, machine classification, image quantization and compression, text retrieval, computational biology and function prediction. On the other hand, in graph theory it is defined the concept of *t-spanner* as a subgraph G' of a graph G that approximates the distances on G with a precision factor t.

A new methodology to solve the metric spaces search problem is presented in this work, which considers a t-spanner $G'(V, E)$ as the metric database representation. In G', the vertices set V corresponds to the metric space objects, and the edges set E corresponds to a reduced selection of the distances between object pairs.

To achieve this goal several t-spanner construction, object insertion and deletion, t-spanner reconstruction, and object retrieval algorithms are proposed. All these algorithms are experimentally evaluated.

Finally, it is conjectured that the essential metric space property for a good t-spanner performance is the existence of element clusters, and enough empirical evidence is given to support this result. This property holds in most real-world metric spaces, so it is expected that t-spanners will have a good behavior in practice. Futhermore, it is shown that t-spanners have a great potential of applications and improvements in the field of metric spaces.

⋆ Este trabajo ha sido financiado en parte por el Núcleo Milenio Centro de Investigación de la Web, Proyecto P01-029-F, Mideplan, Chile; Proyecto CYTED VII.19 RIBIDI y Proyecto MECESUP UCH0109.

1 Introducción

El concepto de *búsqueda aproximada* tiene aplicaciones en muchas áreas de las ciencias de la computación, entre ellas: *bases de datos multimediales* (imágenes, audio, video, etc.), donde el concepto de búsqueda exacta no es aplicable y se buscan objetos similares; *clasificación automática*, donde un nuevo elemento debe ser clasificado de acuerdo a los objetos existentes más cercanos a él; *cuantización y compresión de imágenes*, donde sólo algunos vectores pueden ser representados y los que no, se codifican mediante el representante más cercano; *recuperación de texto*, donde se buscan palabras en una base de datos textual, o se buscan documentos que sean similares a un documento dado; *biología computacional*, donde se busca un ADN o secuencia de proteínas dentro de la base de datos permitiendo algunos errores debidos a las variaciones típicas; y *predicción de funciones*, donde dada una función f, se busca una función g que tenga un comportamiento similar a f de modo de predecir valores futuros de f.

Estas aplicaciones tienen características comunes. Hay un *universo de objetos* $\mathbb{X}$ y una función no negativa $d : \mathbb{X} \times \mathbb{X} \longrightarrow \mathbb{R}^+ \cup \{0\}$ definida entre ellos llamada *distancia* o *métrica*, que satisface las propiedades de positividad ($d(x, y) \geq 0$), simetría ($d(x, y) = d(y, x)$), reflexividad ($d(x, x) = 0$) y la desigualdad triangular ($d(x, y) \leq d(x, z) + d(z, y)$). La métrica modela la "similitud" entre dos objetos (a menor distancia, más similares son). El par $(\mathbb{X}, d)$ es llamado *espacio métrico*.

La *base de datos* en la cual se realizan las búsquedas es un subconjunto finito del universo $\mathbb{U} \subseteq \mathbb{X}$, de tamaño $n = |\mathbb{U}|$, que puede ser preprocesada para construir un índice. Después, dado un nuevo objeto del universo $q \in \mathbb{X}$, se recuperan los elementos de $\mathbb{U}$ similares a q. Interesan dos tipos de consultas de similaridad, que pueden ser resueltas trivialmente con n evaluaciones de distancia:

Consulta de rango $(q, r)_d$: Recupera todos los elementos de $\mathbb{U}$ que estén dentro de la bola de radio r centrada en q. Esto es $\{u \in \mathbb{U} / d(q, u) \leq r\}$.

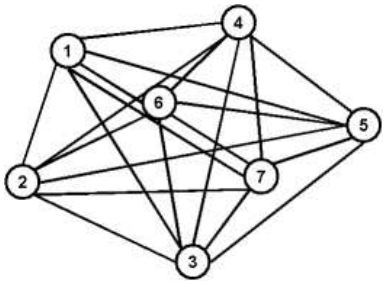
Consulta de k vecinos más cercanos $NN_k(q)_d$: Recupera k elementos de $\mathbb{U}$ más cercanos a q. Esto es, un conjunto $A \subseteq \mathbb{U}$ tal que $|A| = k$ y $\forall\ u \in A,\ v \in \mathbb{U} - A,\ d(q, u) \leq d(q, v)$.

En general, la función d es costosa de calcular (piense por ejemplo en comparar dos huellas dactilares), por lo cual se toma la cantidad de evaluaciones de distancia como la medida de complejidad de la búsqueda. Luego, las técnicas de búsqueda en espacios métricos indexan la base de datos de modo tal de realizar la menor cantidad de evaluaciones de distancia al recuperar objetos.

Por otro lado, en teoría de grafos existe el concepto de *t-spanner*. Sea un grafo conexo no dirigido $G(V, A)$ con una función de costo no negativa $d(e) : A \longrightarrow \mathbb{R}^+ \cup \{0\}$ asignada a sus arcos $e \in A$, y $d_G(u, v)$ el costo del camino más corto entre $u, v \in V$. Un t-spanner de G es un subgrafo conexo, no dirigido $G'(V, E)$, donde $E \subseteq A$, si y sólo si $\forall\ u, v \in V, d_{G'}(u, v) \leq t \cdot d_G(u, v)$ (condición de t-spanner, o t-condición). Los t-spanners tienen diversas aplicaciones, entre ellas: sistemas distribuídos, redes de comunicaciones, arquitectura de máquinas paralelas, robótica y geometría computacional [PS89].

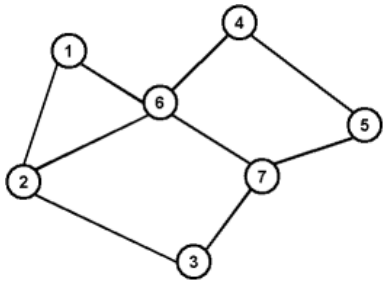
Para ilustrar el concepto de t-spanner consideremos el grafo y matriz de distancias de la Figura 1. Para $t = 1.5$ tenemos un 1.5-spanner que garantiza que las estimaciones de distancia son a lo sumo 1.5 veces las distancias del grafo original; la Tabla 1 muestra los valores máximos permitidos. La Figura 2 muestra el 1.5-spanner con su matriz de distancias; los valores en **negritas** corresponden a distancias de arcos que pertenecen al grafo original, en cambio, los valores en *cursiva* corresponden a las estimaciones de distancia de los arcos no incluidos del grafo original. Note que ningún valor de la matriz de distancias del 1.5-spanner supera su valor respectivo en la Tabla 1.

En este trabajo se utiliza al t-spanner como un índice para una base de datos métrica. Para



	1	2	3	4	5	6	7
1	0	7.5	12.5	6.2	13.5	4	10
2		0	10	12.5	16	7.5	10.5
3			0	12	9	8.5	4
4				0	8.5	5.2	8.3
5					0	9.7	5.5
6						0	6
7							0

Figura 1. Ejemplo de t-spanner. Arriba, grafo inicial. Abajo, matriz de distancias reales.



	1	2	3	4	5	6	7
1	0	**7.5**	*14*	*9.2*	*15.5*	**4**	*10*
2		0	**10**	*16.7*	*23*	**7.5**	*13.5*
3			0	*15.2*	*9.5*	*10*	**4**
4				0	**8.5**	**5.2**	*11.2*
5					0	*11.5*	**5.5**
6						0	**6**
7							0

Figura 2. Ejemplo de t-spanner. Arriba, 1.5-spanner del grafo de la Figura 1. Abajo, matriz de estimación de distancias del 1.5-spanner.

	1	2	3	4	5	6	7
1	0	11.25	18.75	9.3	20.25	6	15
2		0	15	18.75	24	11.25	15.75
3			0	18	13.5	12.75	6
4				0	12.75	7.8	12.45
5					0	14.55	8.25
6						0	9
7							0

Tabla 1. Ejemplo de t-spanner: matriz de estimaciones máximas del grafo de la Figura 1 para $t = 1.5$.

cumplir este objetivo se proponen algoritmos de: construcción de t-spanners, inserción y borrado de objetos en el t-spanner, reconstrucción de t-spanners, y recuperación de objetos desde el t-spanner. Se evaluaron experimentalmente todos los algoritmos propuestos.

El documento original de la tesis es [Par02]. Como resultado de este trabajo se publicaron dos artículos: [NP03] (construcción de t-spanners, ALENEX'03, SIAM Press) y [NPC02] (uso t-spanners para búsqueda en espacios métricos, SPIRE'02, LNCS Springer).

Los contenidos de este resumen son los siguientes. En la Sección 2, se muestran los antecedentes. En la Sección 3, los algoritmos de construcción de t-spanners. En la Sección 4, los algoritmos de mantención y búsqueda en el t-spanner. Los resultados experimentales se muestran en la Sección 5. Por último, en la Sección 6 se muestran las conclusiones y líneas futuras de investigación.

2 Antecedentes

2.1 Soluciones Actuales para Búsqueda en Espacios Métricos

Para espacios métricos generales hay muchas técnicas que indexan el espacio y utilizan la desigualdad triangular para descartar elementos. Sin embargo, estas técnicas de búsqueda pierden efectividad en espacios "difíciles" que se caracterizan por tener un histograma de distancias entre objetos muy concentrado. [CNBYM01] presenta una revisión exhaustiva de estas soluciones, y las clasifica en dos familias: una basada en pivotes y la otra en particiones compactas [JD88]. Una aproximación basada en grafos es [SW90], siendo la más cercana a la propuesta en esta tesis.

Para el caso particular de los espacios vectoriales $\mathbb{R}^D$ con distancia euclidiana, se conocen soluciones eficientes tales como

kd-trees, R-trees, X-trees, etc. [GG98], sin embargo, estas soluciones pierden efectividad para $D \geq 20$. Esta tesis se enfoca a espacios métricos generales, aunque las soluciones presentadas también son eficientes para espacios vectoriales.

Algoritmos Basados en Pivotes. La idea es usar un conjunto de k elementos distinguidos, los "pivotes" $p_1, \ldots, p_k \in \mathbb{U}$ y para cada elemento $u \in \mathbb{U}$ almacenar la distancia hacia los k pivotes $\{d(u,p_1), \ldots, d(u,p_k)\}$. Después, dada la consulta q, se calcula la distancia hacia los k pivotes $\{d(q,p_1), \ldots, d(q,p_k)\}$, y utilizando la desigualdad triangular se descartan los candidatos u tales que $|d(q,p_i) - d(u,p_i)| > r$, sin calcular explícitamente $d(q,u)$. Todos los otros elementos que no puedan ser descartados utilizando esta regla se comparan directamente contra la consulta.

Algoritmos Basados en Particiones Compactas. Esta idea consiste en dividir el espacio en zonas tan compactas como sea posible, y para cada zona almacenar un punto representativo, el "centroide", e información adicional que permita descartar la zona rápidamente al momento de realizar la búsqueda. Luego, cada zona se subdivide recursivamente, produciendo una jerarquía de zonas. Después, dada la consulta q, se calcula la distancia hacia los centroides y se descartan zonas, luego la búsqueda continúa recursivamente en las zonas no descartadas.

Algoritmo AESA. AESA (Approximating Eliminating Search Algorithm) [Vid86] es la técnica de búsqueda aproximada más exitosa. AESA está basado en pivotes y en la noción de que pivotes cercanos a la consulta tienen mejor desempeño en la búsqueda. Para esto toma el extremo $k = n$, lo que permite que cada elemento sea un potencial pivote; de aquí la necesidad de precalcular y almacenar las $n(n-1)/2$ distancias entre todos los objetos. Luego, dado un conjunto de pivotes previos $\{p_1, \ldots, p_{i-1}\}$, el pivote p_i se selecciona como aquel objeto no descartado que minimice la suma de las cotas inferiores de las distancias hacia q, Ecuación (1), que denotaremos $sumLB(u)$.

$$sumLB(u) = \sum_{j=0}^{i-1} \Big| d(p_j,q) - d(p_j,u) \Big| \qquad (1)$$

El descarte de candidatos y la selección de pivotes, continúa hasta revisar todos los objetos. Experimentalmente AESA muestra que tiene un costo casi constante con respecto a n, sin embargo es inaplicable en la práctica dado su requerimiento de memoria $O(n^2)$.

Algoritmo de Shasha y Wang. En [SW90] se emplea un grafo formado por una colección arbitraria de distancias entre pares de objetos que no garantiza ninguna propiedad, dos matrices de tamaño $n \times n$ que computan cotas inferiores y superiores de las distancias, y algoritmos que actualizan las matrices en tiempo $O(n^3)$. Después, las consultas son resueltas utilizando ambas matrices.

La mayor deficiencia de [SW90] es que sólo tiene buen desempeño cuando las distancias se distribuyen uniformemente, lo cual no tiene utilidad práctica. En $\mathbb{R}$, el espacio métrico donde es más fácil buscar, se tiene una distribución de distancias triangular; en espacios métricos generales la distribución de distancias converge asintóticamente a una distribución normal. Ambas distribuciones están lejos de ser uniformes.

2.2 Relación entre Espacios Métricos y t-Spanners

Dado el conjunto de elementos $\mathbb{U}$ se puede considerar un grafo $G(V = \mathbb{U}, A = \mathbb{U} \times \mathbb{U})$, donde $d_G(u,v) = d(u,v)$ es la métrica entre u y v. Un t-spanner $G'(V = \mathbb{U}, E)$ de G permite estimar la distancia entre cada par de objetos con un error conocido t almancenando sólo $|E|$ arcos. Como $E \subseteq \mathbb{U} \times \mathbb{U}$, se tiene que $|E| \leq n^2$. Note que $\forall\, u, v \in \mathbb{U}$ se cumple la Ecuación (2), donde la primera desigualdad se tiene pues G'

es subgrafo de G, y la segunda, corresponde a la t-condición.

$$d(u,v) \;\leq\; d_{t-\text{Spanner}}(u,v) \;\leq\; t \cdot d(u,v) \quad (2)$$

Dado que el histograma de distancias del espacio métrico se estrecha a medida que aumenta la dificultad de buscar por similaridad, valores de $t > 2$ no son apropiados para la aplicación. Además, en virtud de la desigualdad triangular $\forall\ u, v \in \mathbb{U}$ el camino más corto que conecta a u con v en el grafo es precisamente el arco (u, v) y su costo es la distancia medida según la métrica del espacio.

2.3 Estado del Arte de los Algoritmos de Construcción de *t*-Spanners

Hay estudios sobre t-spanners de grafos generales [Epp99,PS89,PU89]. La mayoría proponen algoritmos similares al algoritmo básico presentado en la Sección 2.4. En [ADDJ90,ADD$^+$93] se muestra que esas técnicas producen t-spanners con $n^{1+O(\frac{1}{t-1})}$ arcos para grafos generales de n nodos.

En [PS89] se prueba que dado un grafo compuesto por arcos con peso unitario construir un t-spanner con a lo más m arcos es NP-completo, y sólo se tienen algoritmos que garanticen una cota de aproximación en la cantidad de arcos o peso del t-spanner resultante para el caso $t = 2$ y grafos con arcos de peso unitario [KP94] (donde la cota es $\log \frac{|E|}{|V|}$).

En [Coh98] se presentan algoritmos para grafos generales con $O(n^{1+(2+\varepsilon)(1+\log_n m)/t})$ arcos en tiempo $O(mn^{(2+\varepsilon)(1+\log_n m)/t})$, donde en este caso m se refiere al grafo original y $\varepsilon > 0$. Como $m = |\mathbb{U} \times \mathbb{U}| = \Theta(n^2)$, se tiene un peor caso en tiempo $O(n^5)$. Además, los algoritmos de [Coh98] trabajan para $t \in [2, \log n]$, lo que es inapropiado para espacios métricos.

Para grafos euclidianos, la subclase de grafos donde los vértices son puntos en $\mathbb{R}^D$ con la distancia euclidiana, hay algoritmos para construir t-spanners de $O(n)$ arcos en tiempo $O(n \log^{D-1} n)$ [Epp99,ADDJ90,ADD$^+$93,Kei88,GLN00,RS91]. Lamentablemente, estas técnicas usan la información de las coordenadas, lo que no aplica a los espacios métricos generales.

En [Bar98,CCG$^+$98] se presentan resultados que contemplan aproximaciones probabilísticas utilizando árboles métricos. La idea es construir un conjunto de árboles cuya unión produzca un t-spanner con alta probabilidad. Sin embargo, los valores de t son de la forma $O(\log n \log \log n)$.

De esto surge la necesidad de encontrar algoritmos de construcción de t-spanners apropiados para espacios métricos, vale decir, con $t \leq 2$, para grafos completos y que aprovechen la desigualdad triangular. Interesan algoritmos eficientes tanto en el uso de recursos en la construcción (tiempo de CPU y memoria) como en la calidad del t-spanner resultante (con la menor cantidad de arcos).

2.4 Algoritmo Básico de Construcción de *t*-Spanners

La idea intuitiva para resolver este problema es iterativa. Se comienza con un t-spanner inicial $G'(V = \mathbb{U}, E = \emptyset)$ que contiene todos los vértices y ningún arco, y se inicializan las estimaciones de distancias entre todos los pares de nodos en infinito salvo entre los nodos y sí mismos que son 0 ($d(u,u) = 0$). Luego, los arcos son insertados en orden ascendente de costo y se actualizan las estimaciones de distacia hasta que todas cumplan la t-condición. Note que utilizar primero los arcos pequeños está de acuerdo con la noción geométrica de conectar vecinos cercanos, lo que permite construir caminos a partir de arcos livianos y en total utilizar pocos arcos.

El algoritmo utiliza dos matrices: *real* y *estim*. *real* contiene las distancias reales entre todos los objetos. *estim* contiene las estimaciones de las distancias conseguidas con el t-spanner en construcción. El t-spanner se almacena en una lista de adyacencia.

Los arcos tienen que ser ordenados para luego ser insertados. Un arco se agrega sólo si su estimación actual no cumple la t-condición, y luego de la inserción, se actualizan *todas*

```
t-Spanner0 (Tolerancia t, Vertices 𝕌)

  real ← matriz de distancias reales
  estim ← matriz de estimaciones de distancia
  for u, v ∈ 𝕌 do
    estim(u, v) ← 0 si u = v, ∞ si no
  t-Spanner ← ∅ // estructura de arcos del
                // t-spanner

  for e = (e_u, e_v) ∈ real escogido en orden
                       ascendente do
    if estim(e) > t · real(e) then
      // e está mal t-estimado, es necesario
      // recalcular estim completa
      t-Spanner ← t-Spanner ∪ {e}
      for v_i, v_j ∈ 𝕌 do
        d_1 ← estim(v_i, e_u) + estim(v_j, e_v)
        d_2 ← estim(v_j, e_u) + estim(v_i, e_v)
        estim(v_i, v_j) ← min(estim(v_i, v_j),
                              min(d_1, d_2) + real(e))
```

Figura 3. Algoritmo básico de construcción de t-spanners (t-Spanner 0).

las estimaciones de distancia. El mecanismo de actualización es similar al mecanismo de cálculo de distancias entre todos los nodos del algoritmo de Floyd [Wei95], pero considerando que se insertan arcos en vez de nodos al conjunto de conocidos. La Figura 3 corresponde al algoritmo básico de construcción de t-spanners.

Este algoritmo realiza $O(n^2)$ evaluaciones de distancia al igual que AESA [Vid86]; tiene tiempo de CPU $O(mn^2)$ puesto que por cada arco insertado actualiza la matriz *estim* completa (recuerde que $n = |V|$ y $m = |E|$) y memoria $O(n^2 + m) = O(n^2)$ ya que almacena toda la matriz de distancias estimadas más la estructura del t-spanner. Las deficiencias de este algoritmo son su tiempo de actualización excesivo y que requiere demasiada memoria.

3 Algoritmos para Construcción de t-Spanners

Estos algoritmos de construcción de t-spanners mejoran las deficiencias del algoritmo básico, y consideran valores de $t \leq 2$, grafos métricos completos y la desigualdad del triángulo.

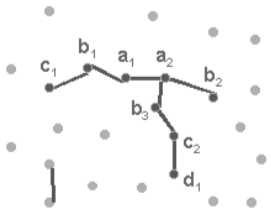


Figura 4. Mecanismo de actualización.

3.1 Algoritmo Básico Optimizado

La idea principal de este algoritmo consiste en acotar la propagación de la actualización de las estimaciones de distancia tras la inserción de un arco. La propagación se detiene cuando un nodo no mejora sus estimaciones o cuando no tiene más vecinos a considerar. La Figura 4 muestra esto gráficamente: la primera actualización sólo modifica los vértices del arco que se inserta, a_1 y a_2; luego, el cómputo se propaga a los vecinos de los nodos a_i: $\{b_1, b_2, b_3\}$; después, a los nodos $\{c_1, c_2\}$ y por último a $\{d_1\}$. Note que sólo es necesario propagar actualizaciones a nodos que mejoran sus estimaciones hacia a_1 o a_2. Se utilizan las matrices *real* y *estim*, y se revisan todos los arcos del grafo ascendentemente. Para cada arco se itera hasta que no se propagen más actualizaciones.

Para controlar las actualizaciones se usan dos conjuntos, *ok* y *check*:

- *ok*: Los nodos que ya tienen actualizadas sus estimaciones de distancia hacia el arco insertado.
- *check*: La adyacencia de *ok*, $check = adyacencia(ok) - ok = \{u \in \mathbb{U} / \exists\ v \in ok, (u, v) \in E\} - ok$. Estos son los nodos que necesitan ser actualizados.

Este algoritmo realiza $O(n^2)$ evaluaciones de distancia; toma tiempo de CPU $O(mk^2)$, donde k es el número de vecinos a chequear cuando inserta un arco, llegando a $O(mn^2)$ en el peor caso tal como el algoritmo básico, pero en promedio es mucho mejor; y necesita memoria $O(n^2 + m) = O(n^2)$. Este algoritmo reduce el tiempo de CPU, aunque aún es elevado, y el requerimiento de memoria es muy

```
t-Spanner1 (Tolerancia t, Vértices 𝕌)

  real ← matriz de distancias reales
  estim ← matriz de estimaciones de distancia
  for u, v ∈ 𝕌 do
    estim(u, v) ← 0 si u = v, ∞ si no
  t-Spanner ← ∅ // estructura de arcos del
                // t-spanner

  for e = (e_u, e_v) ∈ real escogido en orden
                       ascendente do
    if estim(e) > t · real(e) then
      // e está mal t-estimado
      t-Spanner ← t-Spanner ∪ {e}
      ok ← {e_u, e_v}
      check ← adyacencia(ok) - ok
      for c ∈ check do
        // propagando actualizaciones
        if (estim(c, e_v) + real(e) ≤ estim(c, e_u) or
            (estim(c, e_u) + real(e) ≤ estim(c, e_v) then
          for o ∈ ok do
            d_1 ← estim(c, e_u) + estim(o, e_v)
            d_2 ← estim(c, e_v) + estim(o, e_u)
            estim(c, o) ← min(estim(c, o),
                              min(d_1, d_2) + real(e))
          check ← check ∪ (adyacencia(c) − ok)
        ok ← ok ∪ {c}, check ← check − {c}
```

Figura 5. Algoritmo básico optimizado (t-Spanner 1).

alto. La Figura 5 muestra el algoritmo básico optimizado.

La ventaja de este algoritmo es que produce t-spanners de buena calidad, es decir, con pocos arcos. Por esto, los otros algoritmos que insertan varios arcos a la vez, utilizan los resultados de este algoritmo para predecir la cantidad de arcos por nodo. Llamaremos $|E_{t-\text{Spanner }1}(n, d, t)|$ al número esperado de arcos en un espacio métrico de n objetos, función de distancia d y factor de tolerancia t. El modelo se presenta en la Tabla 3, Sección 5.1.

3.2 Algortimo de Inserción Masiva de Arcos

Este algoritmo intenta disminuir tanto el tiempo de CPU como el requerimiento de memoria. El tiempo se reduce actualizando las estimaciones de distancia tras insertar un *conjunto* de arcos con un algoritmo de cálculo de caminos más cortos de tiempo $O(m \log n)$, que llamaremos *Dijkstra*. La memoria se reduce calculando las distancias entre los objetos a medida que se necesitan.

Dado que se insertan varios arcos a la vez (y no uno a uno), el t-spanner resultante necesariamente tiene menor calidad. Los esfuerzos se orientan a minimizar este efecto.

Este algoritmo se desarrolla en tres etapas. En la primera se construye tanto el esqueleto del t-spanner insertando árboles cobertores mínimos (MST, del inglés Minimum Spanning Tree) [Wei95], como la lista global de arcos mal t-estimados, llamada *pendientes*. En la segunda se refina el t-spanner y se corrige la mayoría de los arcos en *pendientes*. En la última se insertan los arcos que queden en *pendientes*. Este algoritmo utiliza dos valores heurísticos H_1 y H_2:

$H1$ determina la cantidad de arcos por nodo y se obtiene a partir del modelo de arcos del algoritmo básico optimizado: $H_1 = |E_{t-\text{Spanner }1}(n, d, t)|/n$. H_1 define el umbral que se utiliza para decidir si insertar o no los arcos que aún están mal t-estimados del nodo que se está revisando. Note que $|E_{t-\text{Spanner }1}(n, d, t)|$ es un predictor optimista del tamaño del t-spanner resultante, luego H_1 puede ser usado como una cota inferior de la cantidad de arcos por nodo.

H_2 determina la capacidad de la lista *pendientes*, y define un criterio para insertar o no MSTs adicionales al t-spanner en construcción, llamado *t-Spanner*. Con experimentos preliminares se determinó que el tamaño más apropiado de *pendientes* es $H_2 = 1.2 \cdot |t\text{-}Spanner|$. Con valores menores a $1.2 \cdot |t\text{-}Spanner|$, el algoritmo toma más tiempo de CPU sin mejorar los resultados, y con valores mayores produce t-spanners de calidad inferior (es decir, con más arcos).

Las etapas de este algoritmo son:

Etapa 1. En la inserción de MSTs, el primer MST sigue el mecanismo usual de Prim [Wei95],

en cambio para los siguientes se utiliza Prim sobre los arcos que aún no han sido insertados.

Los nodos se revisan secuencialmente, para cada nodo se calculan las estimaciones de las distancias con Dijkstra y se insertan a *pendientes* las mal t-estimadas. Cuando el tamaño de *pendientes* sea mayor que H_2 se inserta otro MST. Por otro lado, se remueven de *pendientes* los arcos que hayan mejorado su estimación debido a los nuevos arcos. Además, si el nodo actual tiene menos de $H_1/2$ arcos mal t-estimados, se insertan (con este pequeño conjunto de arcos se corrigen todas las estimaciones de distancia del nodo actual).

Esta etapa continúa hasta que se hayan revisado todos los nodos. La salida de la etapa 1 es el esqueleto del t-spanner guardado en *t-Spanner*, y la lista *pendientes*.

Etapa 2. En la segunda etapa se reduce el tamaño de *pendientes*. Para esto, se revisa la lista de nodos con arcos pendientes, llamada *nodosPendientes*, de mayor a menor arcos pendientes. Para cada nodo, se verifica cuáles arcos han mejorado su t-estimación y cuáles no (la inserción de nuevos arcos puede mejorar estimaciones de otros arcos). De los arcos que siguen mal estimados, se inserta un conjunto de $H_1/4$ arcos de menor costo y se continúa con el siguiente nodo (note que es posible que el nodo continúe con arcos mal t-estimados). Esto entrega un mecanismo para revisar primero los nodos que necesitan mayor atención, sin concentrar todos los esfuerzos en un mismo nodo. Con valores menores a $H_1/4$ el algoritmo toma más tiempo de CPU sin mejorar el resultado y con valores mayores el algoritmo inserta más arcos de los necesarios.

Durante el proceso se consideran dos casos especiales: el primero es que ya se han insertado más de $|V|$ arcos, en donde se regenera *nodosPendientes* y repite el procedimiento. El segundo es que la cantidad de arcos pendientes de un nodo sea tan pequeña ($< H_1/4$) que se insertan.

La condición de salida de esta etapa es $|pendientes| \leq n/2$ (se realizaron experimentos preliminares obteniéndose los mejores resultados con este valor).

Etapa 3. En la tercera etapa se insertan los arcos que quedaron en *pendientes* al t-spanner.

Este algoritmo toma $O(nm)$ evaluaciones de distancia puesto que calcula $O(n^2)$ evaluaciones de distancia por cada MST insertado, tiempo de CPU $O(nm \log m)$ y memoria $O(n+m) = O(m)$. El tiempo de CPU se explica puesto que en la etapa 1 se ejecuta una vez Dijkstra por cada nodo, y en la etapa 2, se insertan grupos de $O(m/n)$ arcos hasta agregar $|pendientes| - n/2 = O(m)$ arcos en total, y debido a H_1 esto se realiza $O(n)$ veces. El uso de memoria se explica puesto que en la etapa 1 la lista *pendientes* nunca es mayor que $O(m)$ ya que crece a lo más en n y tan pronto supera $H_2 = 1.2 \cdot m$ se extraen n arcos para agregar un nuevo MST.

La Figura 6 muestra el algoritmo de inserción masiva de arcos. Este algoritmo reduce tanto el tiempo de CPU como los requerimientos de memoria, pero la cantidad de evaluaciones de distancia es muy alta para la aplicación de interés ($O(nm) \geq O(n^2)$).

3.3 Algoritmo de Inserción Incremental de Nodos

Esta versión pretende disminuir la cantidad de evaluaciones de distancia a $n(n-1)/2$ conservando la idea de amortizar costos en la actualización.

A diferencia de los anteriores, este algoritmo realiza un análisis local de nodos y arcos tomando decisiones sin tener un conocimiento completo del conjunto de arcos. Los nodos se insertan uno tras otro, considerando que para los primeros $i-1$ nodos se tiene un t-spanner bien formado y se agrega el i-ésimo nodo al t-spanner en crecimiento. Debido a que el proceso de inserción sólo analiza localmente el conjunto de arcos, el t-spanner resultante es subóptimo.

Para el i-ésimo nodo, el algoritmo realiza dos operaciones: primero lo conecta al t-spanner utilizando el arco más liviano hacia los

```
t-Spanner2 (Tolerancia t, Vértices 𝕌)

  t-Spanner ← MST // estructura de arcos del t-spanner, inicialmente contiene el primer MST
  pendientes ← ∅ // lista global de arcos mal t-estimados
  H1 ← |E_{t−Spanner 1}(n, d, t)|/n // definición de H1

  Etapa 1: Generación del esqueleto de t-Spanner y pendientes
  for u ∈ 𝕌 do
    if |pendientes| > 1.2 ·|t-Spanner| then // utilizando H2
      t-Spanner ← t-Spanner ∪ MST // construido con los arcos no insertados
    distancia ← Dijkstra(t-Spanner, u) // distancia(v) = d_{t-Spanner}(u, v)
    for v ∈ 𝕌 do
      if distancia(v) ≤ t · d(u, v) then pendientes ← pendientes − {(u, v)}
      else pendientes ← pendientes ∪ {(u, v)}
    if |pendientes(u)| ≤ H1/2 then
      t-Spanner ← t-Spanner ∪ pendientes(u), pendientes ← pendientes − pendientes(u)

  Etapa 2: Eliminación de arcos en pendientes
  while |pendientes| > n/2 do
    nodosPendientes ← nodos ordenados descendentemente según número de arcos pendientes
    for u ∈ nodosPendientes, seleccionado en orden decreciente do
      if ya se han insertado más n arcos then break // caso especial 1
      if |pendientes(u)| < H1/4 then // caso especial 2
        t-Spanner ← t-Spanner ∪ pendientes(u), pendientes ← pendientes − pendientes(u)
      else
        distancia ← Dijkstra(t-Spanner, u)
        for v ∈ pendientes(u) do
          if distancia(v) ≤ t · d(u, v) then pendientes ← pendientes − {(u, v)}
        livianos ← los H1/4 arcos más livianos ∈ pendientes(u)
        t-Spanner ← t-Spanner ∪ livianos, pendientes ← pendientes − livianos

  Etapa 3: t-Spanner ← t-Spanner ∪ pendientes
```

Figura 6. Algoritmo de inserción masiva de arcos (t-Spanner 2), $pendientes(u) = \{e \in pendientes/\ \exists v,\ e = (u, v)\}$.

nodos previos, y luego restaura la t-condición agregando algunos arcos al nodo i-ésimo. Este proceso sigue hasta insertar todos los nodos.

Este algoritmo también utiliza la heurística H_1, recalculándola en cada iteración pues el t-spanner está creciendo. En este algoritmo, se insertan $\delta = H_1/(5 \cdot i)$ nodos a la vez, de modo de reducir el tiempo de CPU y el tamaño del t-spanner (con δ mayores, baja el tiempo de CPU y también la calidad del t-spanner; con δ menores, aumenta el tiempo de CPU pero no la calidad del t-spanner).

Para verificar las estimaciones se utilizó un algoritmo de cálculo de caminos más cortos incremental con propagación límitada, que llamaremos *Dijkstra incremental*. Para esto, el primer cálculo de caminos recibe un arreglo con los costos inicializados en $t \cdot d(v_i, v_j) + \varepsilon$, $\varepsilon > 0$, $j \in [1, i-1]$, (si la distancia hacia el nodo i está mal t-estimada no interesa saber cuánto), para los siguientes cálculos se reutiliza el arreglo de la iteración anterior (no interesa propagar cálculos desde nodos cuya estimación no haya mejorado).

La Figura 7 muestra el algoritmo de inserción incremental de nodos. Este algoritmo toma $O(n^2)$ evaluaciones de distancia, tiempo de CPU $O(nm \log m)$ y memoria $O(n + m) = O(m)$. El tiempo de CPU viene de que cada nodo ejecuta Dijkstra incremental $n/\delta = O(1)$ veces.

3.4 Algoritmo Recursivo

El algoritmo incremental es una buena solución para espacios métricos, pero no considera la cercanía (o lejanía) entre los objetos. Para resolver esto, se podría construir el t-spanner

```
t-Spanner3 (Tolerancia t, Vértices 𝕌)

  t-Spanner ← ∅ // estructura de arcos del
                // t-spanner

  for i ∈ [1, |V|] do
    δ ← |E_{t−Spanner 1}(i, d, t)|/(5 · i) // H1 incremental
    k ← argmin_{j∈[1,i−1]}{(nodo_i, nodo_j)}
    // insertando el arco más liviano
    t-Spanner ← t-Spanner ∪ {(nodo_i, nodo_k)}
    // definiendo el límite de propagación
    distancias ← {(nodo_j, t · d(nodo_i, nodo_j) + ε),
                  j ∈ [1, i − 1]}
    while nodo_i tiene arcos mal t-estimados do
      // Dijkstra incremental
      distancias ← Dijkstra(t-Spanner, nodo_i,
                            distancias)
      pendientes_i ← {(nodo_i, nodo_j), j < i /
            distancias(nodo_j) > t · d(nodo_i, nodo_j)}
      livianos ← el conjunto de los δ arcos más
                 livianos en pendientes_i
      t-Spanner ← t-Spanner ∪ livianos
```

Figura 7. Algoritmo de inserción incremental de nodos (t-Spanner 3).

en un conjunto de objetos cercanos. Con esto se obtiene una solución que divide recursivamente el conjunto de objetos en dos subconjuntos compactos, construye sub-t-spanners en las partes y luego los mezcla.

Para la división inicial del conjunto se toman dos objetos lejanos entre sí, p_1 y p_2, que serán llamados *representantes* y luego se generan los conjuntos de objetos más cercanos a p_1, y más cercanos a p_2 (Figura 8 izquierda). En las divisiones recursivas se reutiliza al representante y se escoge al elemento más lejano a él en su subconjunto como el otro representante. La recursión termina cuando es tamaño del subconjunto es menor que 3.

La mezcla también toma en cuenta la proximidad espacial entre los objetos. Cuando se mezclan los sub-t-spanners $stsp_1$ y $stsp_2$ se tienen dos subconjuntos de nodos, V_1 y V_2, con $|V_1| > |V_2|$ (de no ser así se intercambian). Luego, en $stsp_2$ se escoge el objeto más cercano a p_1 (u_{stsp_2}) y se inserta a $stsp_1$ verificando que las distancias hacia V_1 estén bien t-estimadas. (Esto es equivalente a utilizar el algoritmo incremental para insertar el nodo u_{stsp_2} al t-

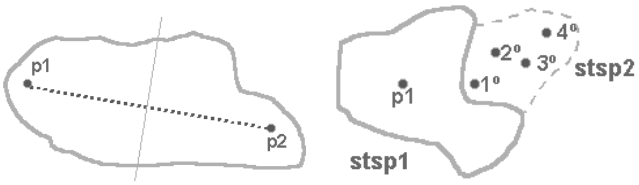


Figura 8. t-Spanner 4: A la izquierda, se selecciona p_1 y p_2 y luego se divide el conjunto. A la derecha, el mezclado toma los elementos de $stsp_2$ y los insertan a $stsp_1$ de acuerdo a la cercanía hacia p_1.

spanner en crecimiento $stsp_1$.) Luego se toma el siguiente objeto más cercano y se repite el proceso hasta que todos los nodos de $stsp_2$ sean insertados a $stsp_1$ (Figura 8 derecha). Note que se conservan los arcos de $stsp_2$ puesto que ya constituyen un t-spanner bien formado para los nodos de V_2.

Este algoritmo también utiliza Dijkstra incremental, pero esta vez sólo interesa limitar la propagación hacia los nodos de $stsp_1$ (hacia $stsp_2$ ya se cumple la t-condición). Luego el arreglo de costos se inicializa con $t \cdot d(v_i, v_j) + \varepsilon$ si $(v_i, v_j) \in V_2 \times V_1$, e ∞ si $(v_i, v_j) \in V_2 \times V_2$, donde ε es una constante positiva pequeña. Para las iteraciones siguientes se reutiliza el arreglo de distancias previamente calculado.

La Figura 9 muestra el algoritmo recursivo y las funciones auxiliares empleadas para construir y mezclar sub-t-spanners. Este algoritmo calcula $O(n^2)$ evaluaciones de distancia, tiempo de CPU $O(nm \log m)$ y memoria $O(n+m) = O(m)$. El costo de dividir los conjuntos es despreciable frente al costo subyacente de la construcción incremental.

3.5 Análisis Comparativo de los Algoritmos de Construcción de t-Spanners

La Tabla 2 muestra las complejidades de los algoritmos de construcción. Se aprecia que los algoritmos apropiados para espacios métricos son el incremental y el recursivo, puesto que tienen la mejor complejidad de tiempo, memoria lineal en el tamaño del t-spanner, y $O(n^2)$ evaluaciones de distancia. Posteriormente, los

```
t-Spanner4 (Tolerancia t, Vértices 𝕌)

  t-Spanner ← ∅ // estructura de arcos del t-spanner
  (p1, p2) ← Selecciona dos objetos lejanos
  (V1, V2) ← Divide 𝕌 según la distancia a (p1, p2)
  stsp1 ← makeSubtSpanner(p1, V1),  stsp2 ← makeSubtSpanner(p2, V2)
  t-Spanner ← mergeSubtSpanner(stsp1, stsp2)

makeSubtSpanner(representante p, Vértices V)
  if |V| = 1 then return t-spanner (nodos ← {p}, arcos ← ∅)
  else if |V| = 2 then return t-spanner (nodos ← V = {v1, v2}, arcos ← {(v1, v2)})
  else // |V| ≥ 3, dividir y mezclar
    p_lejano ← argmax_{v∈V} {d(p, v)}
    (V, V_lejano) ← Divide V según la distancia a (p, p_lejano)
    stsp_p ← makeSubtSpanner(p, V),  stsp_lejano ← makeSubtSpanner(p_lejano, V_lejano)
    return mergeSubtSpanner(stsp_p, stsp_lejano)

mergeSubtSpanner(t-Spanner stsp1, t-Spanner stsp2)
  if |nodos(stsp1)| ≤ |nodos(stsp2)| then stsp1 ⇔ stsp2
  nodos ← nodos(stsp1) ∪ nodos(stsp2)
  arcos ← arcos(stsp1) ∪ arcos(stsp2)
  δ ← |E_{t−Spanner1}(|nodos|, d, t)|/(i · 5) // H1 incremental
  p1 ← representante(stsp1)
  for u ∈ nodos(stsp2) en orden creciente de d(u, p1) do
    // definiendo el límite de propagación hacia stsp1
    for v ∈ nodos(stsp1) do distancias(v) ← t · d(u, v) + ε
    for v ∈ nodos(stsp2) do distancias(v) ← ∞
    while u tiene arcos mal t-estimados hacia stsp1 do
      distancias ← Dijkstra(arcos, u, distancias) // Dijkstra incremental
      pendientes_u ← {(u, v), v ∈ stsp1, distancias(v) > t · d(u, v)}
      livianos ← el conjunto de los δ arcos más livianos en pendientes_u
      arcos ← arcos ∪ livianos
  return t-Spanner (nodos ← nodos, arcos ← arcos)
```

Figura 9. Algoritmo recursivo (t-Spanner 4).

resultados empíricos verifican que el algoritmo recursivo es el que presenta el mejor compromiso de tiempo de CPU y cantidad de arcos.

Adicionalmente, se estudió un algoritmo que combina la metodología de mezclado recursiva con el algoritmo básico optimizado en el caso base de la recursión. Esta versión tiene un análisis muy similar al algoritmo recursivo, y en la práctica, se comportó casi idénticamente al algoritmo recursivo tanto en tiempo de CPU como en la calidad del t-spanner resultante.

	Tiempo de CPU	Memoria	Evaluaciones de distancia
Básico	$O(mn^2)$	$O(n^2)$	$O(n^2)$
Básico optimizado	$O(mk^2)$	$O(n^2)$	$O(n^2)$
Inserción masiva de arcos	$O(nm \log m)$	$O(m)$	$O(nm)$
Incremental	$O(nm \log m)$	$O(m)$	$O(n^2)$
Recursivo	$O(nm \log m)$	$O(m)$	$O(n^2)$

Tabla 2. Comparación de la complejidad de los algoritmos de construcción de t-Spanners. k se refiere al número de nodos deben ser revisados al actualizar estimaciones debido a la inserción de un nuevo arco.

4 Uso de *t*-Spanners para Búsqueda en Espacios Métricos

A continuación se muestran los algoritmos que otorgan la flexibilidad suficiente al t-spanner para utilizarlo como un índice de búsquedas para espacios métricos. Estos algoritmos permiten *actualizar la estructura* y realizar *búsquedas aproximadas* sobre el t-spanner.

4.1 Actualización de la Estructura

La actualización de la estructura consiste en *insertar* y *eliminar* objetos, y *reconstruir* el t-spanner.

Inserción de Objetos al t-Spanner. Suponga que desea insertar un objeto u a un t-spanner $G(V, E)$ para obtener un t-spanner $G_u(V \cup \{u\}, E_u)$. Como *sólo* interesa que las t-estimaciones desde u hacia los elementos de V cumplan la t-condición basta aplicar el algoritmo incremental (Sección 3.3) desde u, recuerde que $G(V, E)$ es un t-spanner bien formado. A diferencia del algoritmo incremental, este algoritmo revisa globalmente las distancias hacia los nodos de V, puesto que en este caso conoce toda la información disponible (el objeto a insertar y el t-spanner).

Borrado de Objetos del t-Spanner. Hay dos opciones, *borrado perezoso* y *borrado efectivo*.

El *borrado perezoso* consiste en marcar el elemento como eliminado. Esto tiene la ventaja de que toma tiempo $O(1)$ y no modifica la estructura del t-spanner, pero tiene la deficiencia de que no libera los recursos ni del objeto borrado, ni de los arcos incidentes a él.

El *borrado efectivo* consiste en eliminar tanto el objeto como los arcos incidentes a él. Para esto se realiza una reconstrucción local del t-spanner en torno al elemento borrado, la que conecta a todos los vecinos del nodo eliminado con arcos "temporales". Note que no todos los arcos temporales son necesarios para conservar la t-condición. La ventaja es que libera los recursos del objeto y sus arcos incidentes, pero puede insertar arcos innecesarios degradando la calidad del t-spanner.

Reconstrucción de t-Spanners. Luego de inserciones y borrados sucesivos la calidad del t-spanner se degrada. Esto motiva la necesidad de un algoritmo de reconstrucción que se aplique periódicamente para mantener la calidad de la estructura.

Reconstruir un t-spanner es equivalente a construir un t-spanner con una selección de arcos ya existente. Esta estrategia permite reutilizar el trabajo previo y realizar una eliminación efectiva de los nodos marcados como borrados y sus arcos incidentes, o de los arcos temporales.

El algoritmo tiene dos etapas, en la primera elimina los nodos y arcos marcados como borrados o temporales, y en la segunda construye el t-spanner utilizando el algoritmo recursivo, a partir del conjunto de nodos y arcos que quedaron tras la eliminación.

4.2 AESA Simulado sobre el t-Spanner

Para simular AESA sobre el t-spanner se debe considerar el efecto de t. Esto significa "extender" el borde superior del rango de exclusión de AESA (con lo cual se pierde capacidad de discriminación) y se modifica el criterio de selección de pivotes (Ecuación (1)).

La condición de exclusión de AESA puede ser reescrita como en la Ecuación (3).

$$d(p, u) < d(p, q) - r \vee d(p, u) > d(p, q) + r \quad (3)$$

Combinando las Ecuaciones (1) y (3) se obtiene la condición de exclusión extendida para el caso de los t-spanners (Ecuaciones (4) y (5)). Dado un pivote p y la consulta q, cualquier objeto u tal que $d(u, q) > r$ satisface la Ecuación (4) ó (5) y se descarta sin evaluar explícitamente $d(u, q)$.

$$d_{t-\text{Spanner}}(p, u) < d(p, q) - r \quad (4)$$

$$d_{t-\text{Spanner}}(p, u) > t \cdot (d(p, q) + r) \quad (5)$$

Por su parte, para compensar el efecto de t, se define $\alpha_t = \frac{2/t+1}{3}$ y se reescribe el criterio para la selección pivotes, seleccionando como pivote aquel elemento u que minimice $sumLB'(u)$, Ecuación (6).

$$sumLB'(u) = \sum_{j=0}^{i-1} \left| d(p_j, q) - \alpha_t \cdot d_{t-\text{Spanner}}(p_j, u) \right| \quad (6)$$

Una vez considerado el efecto de t, se puede simular AESA sobre el t-spanner. Comienza

```
t-AESA (Query q, Radio r, t-Spanner t-Spanner)
  C ← 𝕌
  α_t ← (2/t + 1)/3
  for p ∈ C do sumLB'(c) ← 0

  while C ≠ ∅ do
    p ← argmin_{u∈C}{sumLB'(u)}
    C ← C − {p}
    d_pq ← d(p, q)
    if d_pq ≤ r then Reporta p
    d_{t−Spanner} ← Dijkstra(t-Spanner, p,
                             t(d_pq + r) + ε)
    for u ∈ C do
      if d_{t−Spanner}(u) ∉ [d_pq − r, t(d_pq + r)] then
        C ← C − {u}
      else sumLB'(u) ←
        sumLB'(u) + |d_pq − d_{t−Spanner}(u) · α_t|
```

Figura 10. Algoritmo AESA simulado sobre el t-Spanner (t-AESA).

con el conjunto de candidatos C inicializado con $\mathbb{U}$. Luego se escoge el nodo $p \in C$ que minimize $sumLB'(u)$ y se remueve p de C. Se calcula $d_{pq} = d(p, q)$. Si $d_{pq} \leq r$ se reporta p. Para obtener la distancia entre p y el resto de los elementos se ejecuta Dijkstra desde p, limitando el cálculo a $t(d_{pq} + r)$. Se conservan en C los candidatos que cumplen $d_{pq} - r \leq d_{t-\text{Spanner}}(p, u) \leq t(d(p, q) + r)$. Esto se repite hasta que $C = \emptyset$.

La Figura 10 muestra el algoritmo AESA simulado sobre el t-spanner para resolver la consulta $(q, r)_d$. A partir de esta consulta se pueden resolver consultas de k-vecinos más cercanos $NN_k(q)_d$.

En la práctica, AESA realiza $O(1)$ evaluaciones de distancia y tiempo de CPU $O(n)$ [Vid86] (donde la constante esconde la dimensionalidad del espacio). Sin embargo, la simulación debe considerar el tiempo extra debido al cálculo de caminos de costo mínimo. Con la versión utilizada en la tesis, el tiempo de CPU utilizado es $O(n_p \cdot m \log n)$, donde n_p es la cantidad de nodos utilizados como pivote y $m = |E| = n^{1+\alpha}$, $\alpha \in [0, 1]$. Luego el costo de cada consulta corresponde al costo de AESA multiplicado por $O(n^\alpha \log n)$, $\alpha \in [0, 1]$. Se recuerda que en estas aplicaciones el costo de computar la distancia d domina sobre cualquier otro costo de CPU.

5 Resultados Experimentales

Se realizaron experimentos de construcción y búsqueda en espacios vectoriales gaussianos, espacios de strings y espacios de documentos (los dos últimos son de interés en Recuperación de la Información [BYRN99]). Los experimentos fueron realizados en un Intel Pentium IV de 2.0 GHz, con 512 MB de RAM y disco duro local.

Por economía se denominará t-Spanner 1 al algoritmo básico optimizado, t-Spanner 2 al algoritmo de inserción masiva de arcos, t-Spanner 3 al algoritmo incremental, t-Spanner 4 al algoritmo recursivo y t-AESA a AESA simulado sobre el t-spanner.

Los experimentos de construcción comparan t-Spanner 1, 2, 3 y 4 para determinar cuál es más apropiado para espacios métricos.

Los experimentos de búsqueda utilizan t-spanners con factores de precisión $t \in [1.4, 2.0]$ y se compara contra AESA y un algoritmo basado en pivotes escogidos al azar, variando la cantidad de pivotes. Para cada valor de t, se mide el tamaño del t-spanner resultante y se construye un índice basado en pivotes que use la misma cantidad de memoria. En los casos que se tengan más pivotes que elementos a descartar, se detiene el uso de pivotes. También, de ser necesario, se muestran resultados con menos pivotes hasta alcanzar el óptimo.

5.1 Espacio Métrico $\mathbb{R}^D$ Gaussiano con Distancia Euclidiana

Es sabido que los espacios métricos reales presentan regiones llamadas *clusters* (en español racimos o grupos), en donde se concentran los objetos del espacio. Con el espacio vectorial gaussiano se pretende simular un espacio métrico real. Se consideraron tres valores de desviación estándar ($\sigma = 0.1$, 0.3 y 0.5) para estudiar el desempeño con clusters concentrados o dispersos. El espacio considera 256 clusters distribuidos uniformemente y coordenadas en

20 dimensiones en el rango [-1.0, 1.0]. No se utilizó el hecho de que el espacio tiene coordenadas, sino que se empleó como si los puntos fueran objetos abstractos de un espacio métrico desconocido.

En esta sección se realiza una comparación experimental de los algoritmos de construcción propuestos en la tesis para escoger aquel que tenga mejor desempeño para realizar las pruebas masivas de búsqueda. Por otro lado, se verifica empíricamente la conjetura de que el t-spanner obtiene mayores beneficios de la existencia de clusters en el espacio que estructuras alternativas.

Construcción. La Figura 11 muestra los resultados de los cuatro algoritmos de construcción para $n \in [200; 2{,}000]$, $t = 1.4$ y 1.8, y $\sigma = 0.1$ y 0.5. Como se aprecia, los algoritmos producen t-spanners de calidad similar, siendo t-Spanner 1 consistentemente mejor. Note que t-Spanner 2 tiene el peor desempeño en arcos para $\sigma = 0.1$, esto porque en su etapa 1, por cada MST inserta demasiados arcos intra-clusters y pocos entre-clusters, luego hay demasiados arcos que no aportan a satisfacer la t-condición; esto se revierte para $\sigma = 0.5$ donde llega a ser el segundo mejor algoritmo.

Con respecto al tiempo de construcción se tienen diferencias importantes. t-Spannner 1 toma un tiempo de CPU altísimo. Por otro lado, t-Spanner 2 es bastante costoso frente a t-Spanner 3 y 4 debido a la gran cantidad de evaluaciones de distancia que efectúa. Sin embargo, a diferencia de los otros algoritmos, t-Spanner 2 mejora su desempeño cuando los clusters se hacen más dispersos.

Los algoritmos t-Spanner 3 y 4 tienen mediciones muy parecidas para tiempo de CPU y arcos, siendo los algoritmos más rápidos. Sin embargo, t-Spanner 4 usualmente produce t-spanners de mejor calidad gracias a su análisis global de arcos. En efecto, t-Spanner 4 produce 1.5-spanners cuyo tamaño es de un 5% a un 15% del grafo completo.

Tras este análisis se selecciona al algoritmo recursivo como el mejor algoritmo para indexar la base de datos métrica con un t-spanner, pues es el que tiene el mejor compromiso entre tiempo de CPU y tamaño del t-spanner producido. Este resultado también se verifica en los espacios de strings y de documentos.

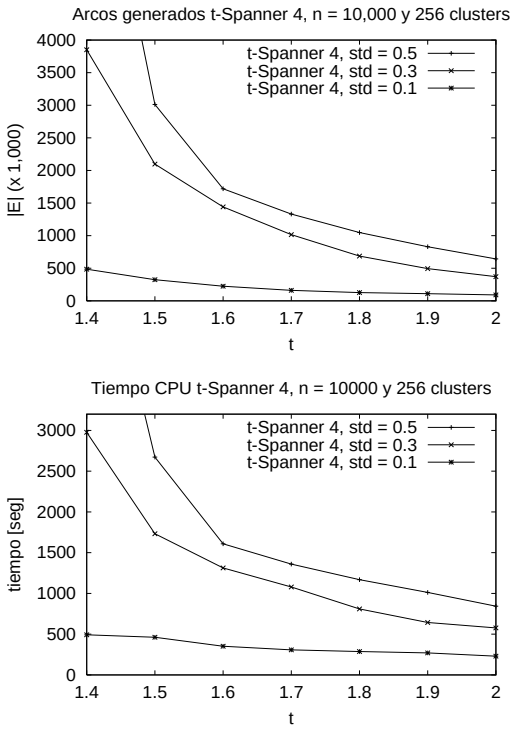


Figura 12. Construcción de t-spanners en espacio vectorial gaussiano de 10,000 nodos, 256 clusters y $\sigma = 0.1$, 0.3 y 0.5 utilizando t-Spanner 4 variando t. Arriba, arcos generados. Abajo, tiempo de construcción. Para $\sigma = 0.5$, 1.4-Spanner 4 alcanza 8.2 millones de arcos (arriba) y 6,200 segundos (abajo).

La Figura 12 muestra los resultados de construcción de t-spanners para 10,000 nodos con el algoritmo recursivo. Note que tanto la cantidad de arcos como el tiempo de CPU disminuyen a medida que σ se hace más pequeño. Esto se debe a que el mecanismo de división en subconjuntos compactos detecta clusters por construcción, y el mecanismo de mezclado puede hacer buenas aproximaciones de las distancias de un cluster a otro utilizando pocos arcos. Luego, el t-spanner generado se adapta y aprovecha la existencia de clusters.

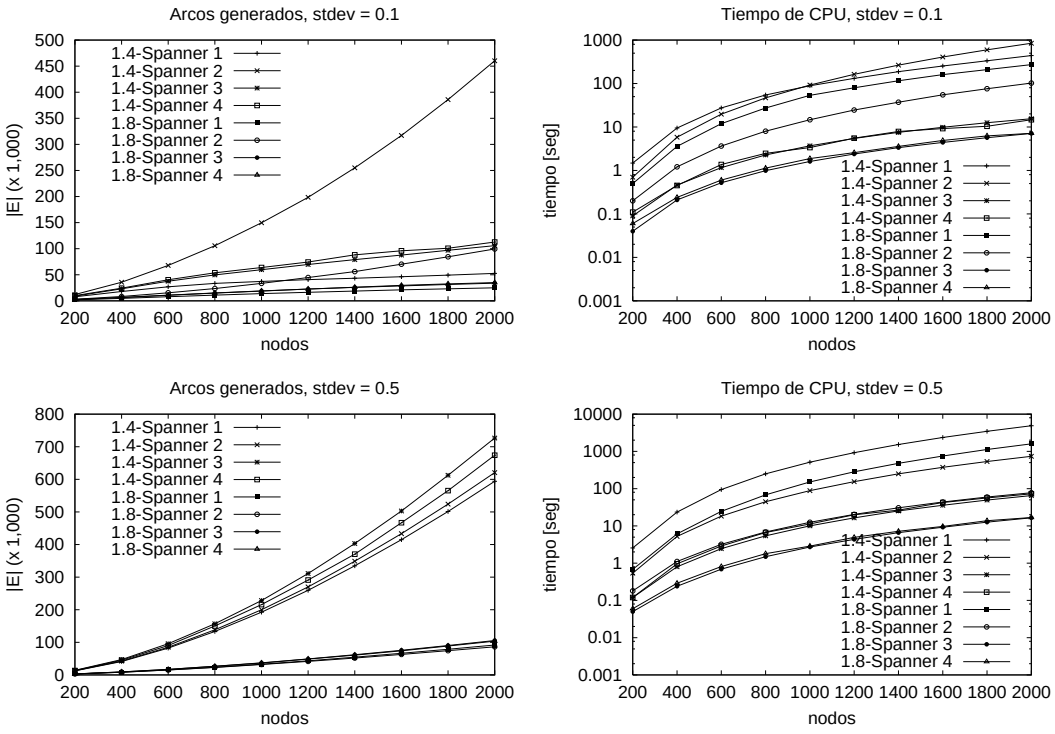


Figura 11. Construcción de t-spanners en un espacio gaussiano de 2,000 nodos, 256 clusters y $\sigma = 0.1$ y 0.5 en función del número de nodos. A la izquierda, arcos generados (calidad del t-spanner). A la derecha, tiempo de construcción.

La Tabla 3 muestra los ajustes de mínimos cuadrados usando modelos de la forma $|E| = an^{1+\frac{b}{t-1}}$ y $tiempo = an^{2+\frac{b}{t-1}}$ obtenidos de [ADDJ90,ADD$^+$93]. Como se ve, el tamaño de los t-spanners es levemente superlineal y el tiempo es levemente supercuadrático. Esto muestra que los algoritmos de construcción representan una mejora importante sobre el estado del arte actual.

Búsqueda. La Figura 13 muestra los resultados de búsqueda sobre un conjunto de 10,000 objetos gaussianos indexados con t-Spanner 4. Para $\sigma = 0.1$, 0.3 y 0.5, un 1.4-spanner indexa la base de datos utilizando respectivamente el 0.97%, 7.70% y 16.32% de la memoria de AESA. Se aprecia que a menor valor de t mejor el desempeño de t-AESA tanto para recuperar 1 como 10 objetos.

Por otro lado, el diámetro de los clusters también influye en el desempeño de t-AESA. Para clusters concentrados los resultados son competitivos frente a AESA y mejores que con los pivotes, pues t-spanner se adapta y aprovecha la presencia de clusters, por ejemplo: para $\sigma = 0.1$, 1.4-AESA recupera respectivamente 1 y 10 objetos con 1.05 y 1.04 veces las evaluaciones de AESA, por su parte los pivotes utilizan un exceso de 1.35 y 1.55 veces para 1 y 10 objetos; para $\sigma = 0.3$, 1.4-AESA recupera 1 y 10 objetos con 1.53 y 2.72 veces las evaluaciones de AESA, y los pivotes utilizan 5.37 y 5.63 veces para 1 y 10 objetos. Esto se revierte para cluster dispersos, donde los objetos se distribuyen casi uniformemente y el t-spanner pierde capacidad de discriminación de objetos en la búsqueda, por ejemplo, con $\sigma = 0.5$ se recupera 1 y 10 objetos con 2.49 y 2.06 veces las evaluaciones de distancia de AESA y el óptimo de los pivotes utiliza 1.62 y 1.40 veces, respectivamente.

De este modo se verifica experimentalmente la conjetura de que los t-spanners aprovechan los clusters que naturalmente se forman en

σ		Básico optimizado	Inserción masiva de arcos	Incremental	Recursivo
0.1	Tiempo CPU [μseg]	$17.8\ n^{2+\frac{0.09}{t-1}}$	$1.67\ n^{2+\frac{0.24}{t-1}}$	$0.670\ n^{2+\frac{0.10}{t-1}}$	$0.909\ n^{2+\frac{0.08}{t-1}}$
0.1	Arcos [arcos]	$5.76\ n^{1+\frac{0.10}{t-1}}$	$6.50\ n^{1+\frac{0.18}{t-1}}$	$6.17\ n^{1+\frac{0.13}{t-1}}$	$5.77\ n^{1+\frac{0.14}{t-1}}$
0.3	Tiempo CPU [μseg]	$25.0\ n^{2+\frac{0.16}{t-1}}$	$1.52\ n^{2+\frac{0.22}{t-1}}$	$0.771\ n^{2+\frac{0.13}{t-1}}$	$0.865\ n^{2+\frac{0.13}{t-1}}$
0.3	Arcos [arcos]	$5.69\ n^{1+\frac{0.18}{t-1}}$	$5.41\ n^{1+\frac{0.19}{t-1}}$	$6.52\ n^{1+\frac{0.19}{t-1}}$	$6.50\ n^{1+\frac{0.18}{t-1}}$
0.5	Tiempo CPU [μseg]	$21.0\ n^{2+\frac{0.19}{t-1}}$	$1.33\ n^{2+\frac{0.25}{t-1}}$	$0.587\ n^{2+\frac{0.17}{t-1}}$	$0.650\ n^{2+\frac{0.17}{t-1}}$
0.5	Arcos [arcos]	$4.89\ n^{1+\frac{0.21}{t-1}}$	$4.50\ n^{1+\frac{0.22}{t-1}}$	$5.20\ n^{1+\frac{0.22}{t-1}}$	$5.37\ n^{1+\frac{0.21}{t-1}}$

Tabla 3. Complejidades empíricas para los algoritmos de construcción en fución de n y t.

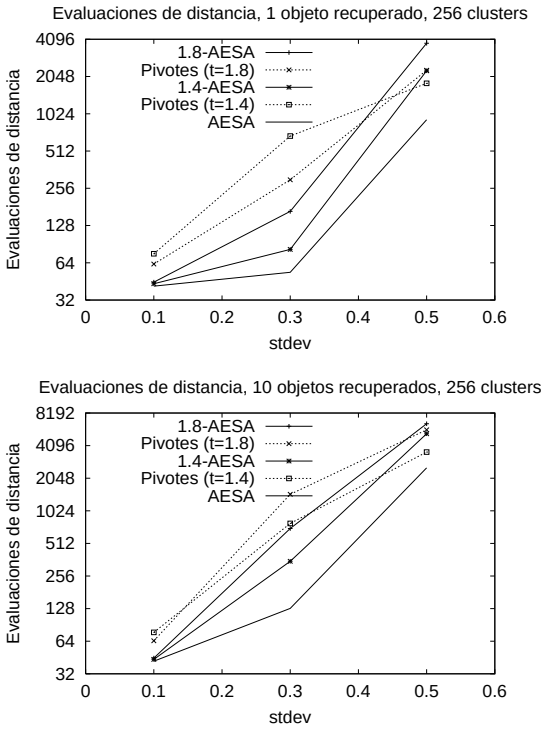


Figura 13. Consultas por rango en espacios vectoriales gaussianos. Arriba, en promedio se recupera 1 objeto por consulta. Abajo, en promedio se recuperan 10 objetos por consulta.

un espacio métrico real, y mientras más concentrados los clusters, mejor el desempeño de los t-spanners. Es sabido que todos los algoritmos de búsqueda mejoran cuando hay clusters, pero los t-spanners mejoran más que, por ejemplo, los pivotes.

5.2 Espacio Métrico de los Strings con Distancia de Edición

El espacio métrico de los strings con la distancia de edición no posee coordenadas. La distancia de edición es una función discreta que mide el mínimo número de inserciones, borrados y reemplazos de caracteres de modo tal de hacer que dos strings sean iguales [NR02]. Se consideró un conjunto de 24,000 palabras de un diccionario inglés. Debido a la gran cantidad de objetos las pruebas de construcción se hicieron con t-Spanner 3 y 4 variando la cantidad de nodos. Las pruebas de búsqueda consideran radios de recuperación $r = 1$ y 2, variando t.

Construcción. La Figura 14 muestra los resultados de construcción. Como se aprecia, el número de arcos es levemente superlineal ($8.03\ n^{1+\frac{0.16}{t-1}}$ para t-Spanner 3 y $8.45\ n^{1+\frac{0.15}{t-1}}$ para t-Spanner 4), y el tiempo de CPU es levemente supercuadrático ($1.46\ n^{2+\frac{0.10}{t-1}}\ \mu seg$ para t-Spanner 3 y $1.67\ n^{1+\frac{0.09}{t-1}}\ \mu seg$ para t-Spanner 4). Se verifica que t-Spanner 4, es generalmente mejor que t-Spanner 3 en tiempo y memoria. Note que el grafo completo de 24,000 nodos tiene aproximadamente 288 millones de arcos y un 1.4-spanner tiene sólo 8.35 millones de arcos (2.90% del grafo completo).

Búsqueda. Para radio 1 (Figura 15 arriba), con $t = 2.0$ se comporta mejor el algoritmo basado en pivotes, sin embargo, con valores de $t \leq 1.7$ (lo que produce un índice que necesita más memoria), t-AESA responde sistemáticamente mejor, llegando a ser muy competitivo para

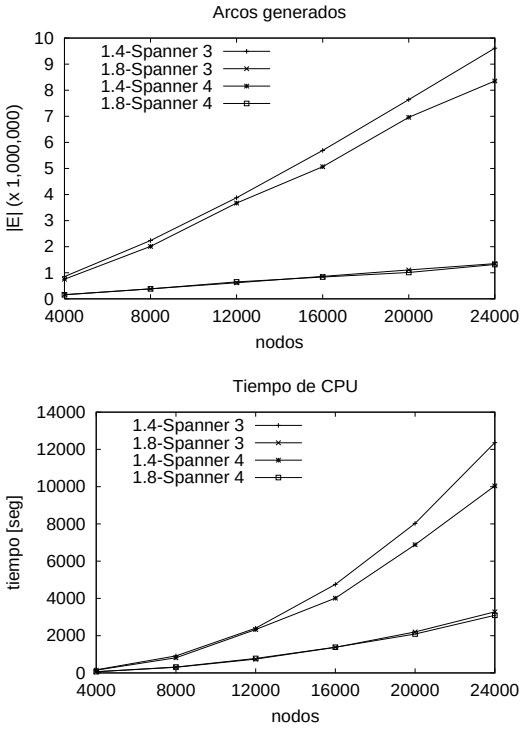


Figura 14. Construcción de t-Spanners en espacio de strings variando t, $n = 24{,}000$. Arriba, arcos generados. Abajo, tiempo de CPU.

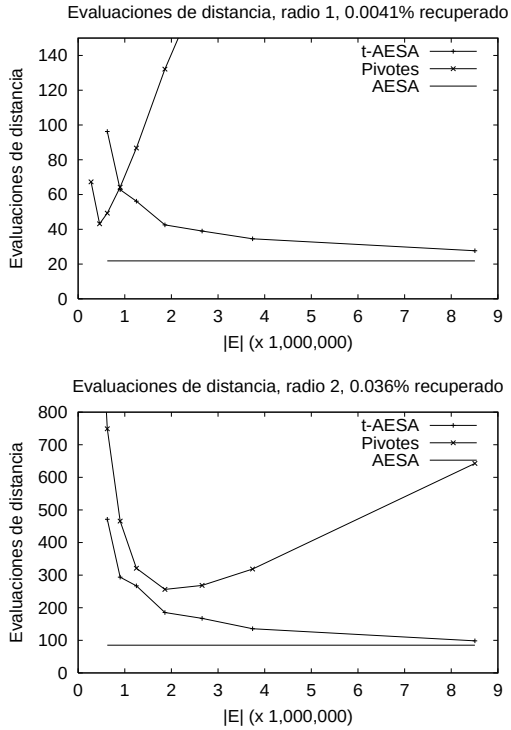


Figura 15. Consultas por rango en espacios de strings. Arriba, radio $r = 1$, el algoritmo basado en pivotes alcanza 540 evaluaciones de distancia para 8.5 millones de arcos ($t = 1.4$). Abajo, radio $r = 2$.

$t = 1.4$ donde realiza sólo un 27% más de evaluaciones de distancia que AESA. En cambio, se aprecia que el algoritmo basado en pivotes no mejora los resultados en la medida de que dispone más memoria.

Para radio 2 (Figura 15 abajo), t-AESA tiene mejor desempeño que los pivotes para todo $t \in [1.4, 2.0]$, con sólo un 16% de evaluaciones de distancia adicionales para $t = 1.4$. También se aprecia que el algoritmo basado en pivotes no mejora sus resultados a medida que dispone de más memoria, en cambio, el t-spanner *siempre* mejora a medida que t disminuye.

Para entender estos resultados tan favorables se debe contemplar que, junto con el hecho de que el espacio de strings tiene clusters de radio pequeño, la distancia de edición es una función discreta, lo que en la práctica colabora en la capacidad de discriminación de t-AESA.

5.3 Espacio Métrico de los Documentos con Distancia Coseno

Es usual representar a los documentos como si fuesen vectores en un espacio de alta dimensionalidad, donde cada dimensión corresponde a un término del vocabulario de la colección completa de documentos. La distancia coseno mide el coseno del ángulo formado entre dos vectores (producto interno) [BYRN99] y es una función muy costosa de calcular (toma varios milisegundos). Este espacio tiene un histograma de distancias muy concentrado y se considera intratable.

Se usa una base de 1,200 documentos. Las pruebas de construcción comparan t-Spanner 1, 3 y 4 (se excluye t-Spanner 2 pues necesita $O(nm) > O(n^2)$ evaluaciones de distancia), con $t \in [1.4, 2.0]$. Las pruebas de búsqueda recuperan 1 y 10 documentos en promedio por consulta, con $t \in [1.4, 2.0]$.

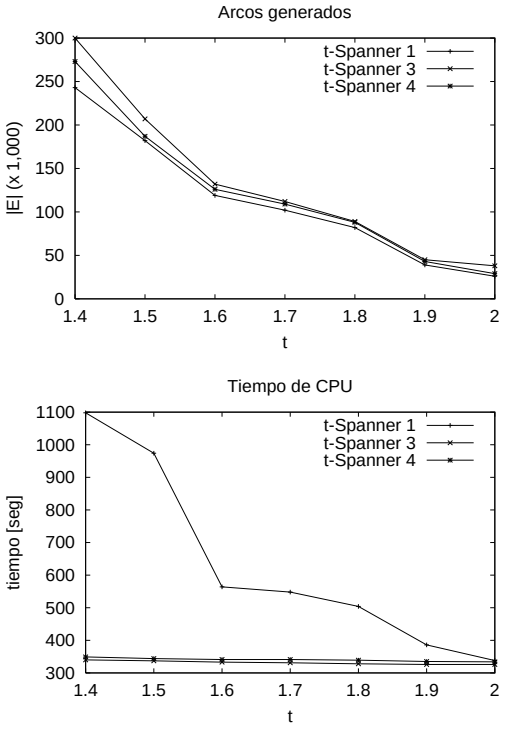


Figura 16. Construcción de t-Spanner en espacio de documentos variando t, n = 1,200. Arriba, arcos generados. Abajo, tiempo de CPU.

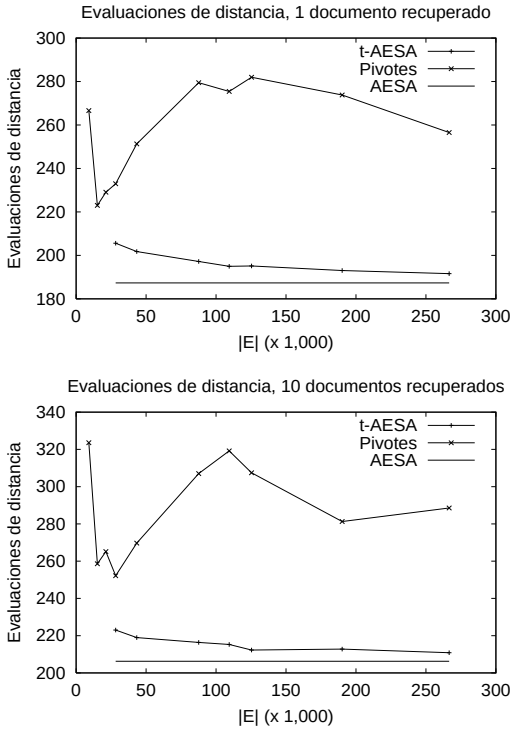


Figura 17. Consultas por rango en espacios de documentos. Arriba, recupera en promedio 1 documento por consulta, radio r = 0.1325. Abajo, recupera en promedio 10 documentos por consulta, radio r = 0.167.

Construcción. La Figura 16 muestra que los tres algoritmos producen t-spanners de calidad similar. La distancia coseno es el término dominante del tiempo de construcción, esto explica que t-Spanner 3 y 4 tomen prácticamente el mismo tiempo para $t \in [1.4, 2.0]$. Por otro lado t-Spanner 1 es mucho más costoso que los otros. El grafo completo de 1,200 nodos tiene al rededor de 720 mil arcos y un 2.0-spanner tiene sólo 28 mil (3.83% del grafo completo).

Búsqueda. Para todo el rango estudiado, t-AESA se comporta mejor que el algoritmo basado en pivotes, y es extremadamente competitivo frente a AESA (Figura 17). Para $t = 2.0$, con radio $r = 0.1325$, realiza 9% más de las evaluaciones de distancia de AESA para recuperar 1 documento, y con radio $r = 0.167$ sólo un 8% adicional para recuperar 10 documentos.

6 Conclusiones

El objetivo de esta tesis consistió en abordar el problema de búsqueda en espacios métricos utilizando índices y algoritmos de búsqueda basados en *t-spanners*. A continuación se resumen los principales logros de la tesis y se analiza su impacto. Parte de estos resultados fueron publicados en [NP03] y [NPC02].

Construcción de *t*-spanners. Se proponen algoritmos que construyen t-spanners de buena calidad en tiempo razonable. Se obtuvieron tiempos empíricos de la forma $C_t \cdot n^{2+\frac{0.1...0.2}{t-1}}$ y número de arcos de la forma $C_e \cdot n^{1+\frac{0.1...0.2}{t-1}}$. Note que calcular el árbol cobertor mínimo requiere tiempo $O(n^2)$ y calcular todas las distancias en un grafo general requiere tiempo $O(n^3)$. Comparado con los algoritmos existentes, esta contribución representa un aporte significativo al estado actual del arte. Por último, se

selecciona al algoritmo recursivo como el más apropiado para espacios métricos.

Actualización de la estructura. Se proponen mecanismos de inserción y borrado de elementos y de reconstrucción de la estructura. La combinación de estos mecanismos permite mantener actualizada la estructura de la base de datos preservando su calidad.

Recuperación de objetos. Se propone un algoritmo para resolver consultas por rango sobre el t-spanner. En esta metodología, mientras más memoria se tenga disponible, mejor es el desempeño de la búsqueda (note que los métodos basados en pivotes no presentan esta característica).

Comparación de t-spanners con otras metodologías. Los resultados experimentales muestran que la metodología basada en t-spanners es muy competitiva en comparación con otras técnicas de búsqueda en espacios métricos. Por ejemplo, en el espacio de los documentos con un 2.0-spanner, que utiliza sólo un 3.84% de la memoria de AESA, se resuelven las consultas por rango utilizando sólo 1.09 veces las evaluaciones de distancia de AESA, y es mucho mejor que el algoritmo basado en pivotes. Similar resultado se tiene para el espacio de los strings.

Ventajas de la metodología basada en t-spanners.

- Los t-spanners presentan un muy buen comportamiento en espacios métricos compuestos por datos obtenidos del mundo real. En particular, permiten un excelente compromiso entre tiempo y memoria como alternativa a AESA.
- Bajo la presencia de clusters, el t-spanner se adapta naturalmente a la distribución espacial de los datos, permitiendo obtener mejores resultados tanto en la fase de construcción como en la de búsqueda (en particular, mejor que con algoritmos basados en pivotes). Se proporciona evidencia empírica de que este buen comportamiento se mantiene en cualquier espacio que presente clusters. La mayoría de los espacios métricos de la vida real presentan clusters, por lo que se espera que los t-spanners se comporten bien en la práctica.
- Uno de los problemas de esta metodología, es que requiere de un mayor esfuerzo de cálculo para descartar candidatos al momento de resolver las consultas de búsqueda aproximada. Sin embargo, cuando el costo de la evaluación de distancia es efectivamente elevado (por ejemplo, la distancia coseno), se hace despreciable el esfuerzo de cómputo de las estimaciones de distancia, siendo éste un caso donde realmente se notan las virtudes de esta metodología.

Trabajo futuro.

- En los espacios métricos es especialmente útil tener una buena estimación de las distancias cortas puesto que ésto mejora el rendimiento de la búsqueda. De aquí nace la idea de considerar una función $t(u, v)$ tal que a menor distancia entre los nodos menor sea el valor de t, y desarrollar algoritmos de construcción y búsqueda que usen esta función $t(u, v)$.
- Se podría mejorar los algoritmos de búsqueda utilizando el t-spanner como una estructura de navegación entre los objetos. De esta manera, la búsqueda se divide en una fase de acercamiento hacia la consulta, y otra de descarte de los candidatos que estén fuera del rango de exclusión.
- Desarrollar una estructura de datos métrica completamente dinámica, es decir, que permita insertar y borrar objetos sin tener que realizar reconstrucciones periódicas de la estructura, conservando un buen desempeño en las operaciones de búsqueda.

Agradecimientos

A mis profesores Edgar Chávez y Luis Dissett por sus valiosos aportes a mi trabajo en la tesis.

Referencias

[ADD+93] I. Althöfer, G. Das, D. Dobkin, D. Joseph, and J. Soares. On sparse spanners of weighted graphs. *Discrete Computational Geometry*, 9:81–100, 1993.

[ADDJ90] I. Althöfer, G. Das, D. Dobkin, and D. Joseph. Generating sparse spanners for weighted graphs. In *Proc. 2nd Scandinavian Workshop on Algorithm Theory (SWAT'90)*, LNCS 447, pages 26–37, 1990.

[Bar98] Y. Bartal. On approximating arbitrary metrics by tree metrics. In *Proc. 30th Symp. on the Theory of Computing (STOC'98)*, pages 161–168, 1998.

[BYRN99] R. A. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley, 1999.

[CCG+98] M. Charikar, C. Chekuri, A. Goel, S. Guha, and S. Plotkin. Approximating a finite metric by a small number of tree metrics. In *Proc. 39th Symp. on Foundations of Computer Science (FOCS'98)*, pages 379–388, 1998.

[CNBYM01] E. Chávez, G. Navarro, R. Baeza-Yates, and J.L. Marroquín. Proximity searching in metric spaces. *ACM Computing Surveys*, 33(3):273–321, September 2001.

[Coh98] E. Cohen. Fast algorithms for constructing t- spanners and paths with stretch t. *SIAM Journal on Computing*, 28:210–236, 1998.

[Epp99] D. Eppstein. Spanning trees and spanners. In *Handbook of Computational Geometry*, pages 425–461. Elsevier, 1999.

[GG98] V. Gaede and O. Günther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, 1998.

[GLN00] J. Gudmundsson, C. Levcopoulos, and G. Narasimhan. Improved greedy algorithms for constructing sparse geometric spanners. In *Proc. 7th Scandinavian Workshop Algorithm Theory (SWAT'00)*, LNCS 1851, pages 314–327, 2000.

[JD88] A.K. Jain and R.C. Dubes. *Algorithms For Clustering Data*. Prentice-Hall, Englewood Cliffs, 1988.

[Kei88] J.M. Keil. Approximating the complete Euclidean graph. In *Proc. 1st Scandinavian Workshop in Algorithm Theory (SWAT'88)*, LNCS 318, pages 208–213, 1988.

[KP94] G. Kortsarz and D. Peleg. Generating sparse 2-spanners. *Journal of Algorithms*, 17(2):222–236, 1994.

[NP03] G. Navarro and R. Paredes. Practical construction of metric t-spanners. In *Proc. 5th Workshop on Algorithm Engineering and Experiments (ALENEX'03)*, pages 69–81. SIAM Press, 2003.

[NPC02] G. Navarro, R. Paredes, and E. Chávez. t-Spanners as a data structure for metric space searching. In *Proc. 9th Intl. Symp. on String Processing and Information Retrieval (SPIRE'02)*, LNCS 2476, pages 298–309. Springer, 2002.

[NR02] G. Navarro and M. Raffinot. *Flexible Pattern Matching in Strings – Practical on-line search algorithms for texts and biological sequences*. Cambridge University Press, 2002.

[Par02] R. Paredes. *Uso de t-spanners para búsqueda en espacios métricos*. Tesis de Magíster, Universidad de Chile, 2002. `www.dcc.uchile.cl/~raparede/th.ps.gz`.

[PS89] D. Peleg and A. Schaffer. Graph spanners. *Journal of Graph Theory*, 13(1):99–116, 1989.

[PU89] D. Peleg and J. Ullman. An optimal synchronizer for the hypercube. *SIAM J. on Computing*, 18:740–747, 1989.

[RS91] J. Ruppert and R. Seidel. Approximating the d-dimensional complete Euclidean graph. In *3rd Canadian Conference on Computational Geometry*, pages 207–210, 1991.

[SW90] D. Shasha and T. Wang. New techniques for best-match retrieval. *ACM Trans. on Information Systems*, 8(2):140–158, 1990.

[Vid86] E. Vidal. An algorithm for finding nearest neighbors in (approximately) constant time. *Pattern Recognition Letters (PRL'86)*, 4:145–157, 1986.

[Wei95] M. A. Weiss. *Estructuras de datos y algoritmos*. Addison-Wesley Iberoamericana, 1995.