

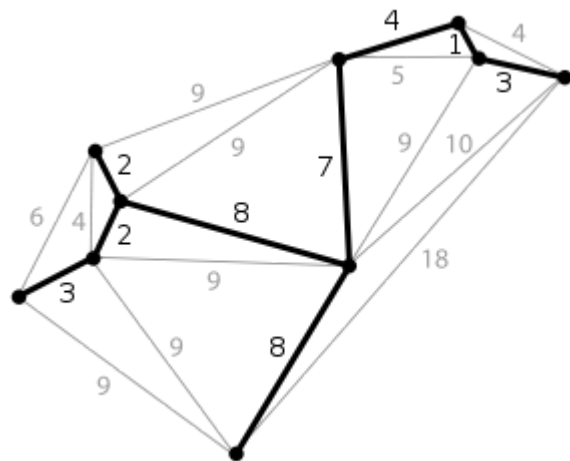
ALUNO : JOSÉ RIBAMAR DURAND RODRIGUES JUNIOR

RELATÓRIO ESTRUTURA DE DADOS

INTRODUÇÃO

O algoritmo de Kruskal é um algoritmo que pega um grafo como entrada e encontra o subconjunto das arestas daquele gráfico que formam uma árvore que inclua todos os vértices e que tenha a soma mínima de pesos entre todas as árvores que podem ser formadas a partir do grafo, uma árvore geradora mínima.

Uma árvore de extensão mínima (também conhecida como árvore de extensão de peso mínimo ou árvore geradora mínima) é então uma árvore de extensão com peso menor ou igual a cada uma das outras árvores de extensão possíveis. Generalizando mais, qualquer grafo não direcional (não necessariamente conectado) tem uma floresta de árvores mínimas, que é uma união de árvores de extensão mínimas de cada uma de suas componentes conexas.



Um exemplo de uso de uma árvore de extensão mínima seria a instalação de fibras óticas num campus de uma faculdade. Cada trecho de fibra ótica entre os prédios possui um custo associado (isto é, o custo da fibra, somado ao custo da instalação da fibra, mão de obra, etc). Com esses dados em mãos (os prédios e os custos de cada trecho de fibra ótica entre todos os prédios), podemos construir uma árvore de extensão que nos diria um jeito de conectarmos todos os prédios sem redundância. Uma árvore geradora mínima desse grafo nos daria uma árvore com o menor custo para fazer essa ligação.

O Kruskal se enquadra em uma classe de algoritmos chamados algoritmos gananciosos, que encontram o ideal local na esperança de encontrar um ótimo global. Começamos pelas arestas com o menor peso e continuamos adicionando arestas até alcançarmos nossa meta.

As etapas para implementar o algoritmo de Kruskal são as seguintes:

1. Ordene todas as arestas com base no peso
2. Pegue a aresta com o menor peso e adicione-a à árvore de abrangência. Se adicionar a aresta criou um ciclo, rejeite-a.
3. Continue adicionando arestas até atingirmos todos os vértices.

O Kruskal precisa de um algoritmo de ordenação no primeiro passo da implementação, foram testados 9 algoritmos:

INSERTSORT

A ordenação por inserção itera , consumindo um elemento de entrada a cada repetição e aumentando uma lista de saídas ordenadas. A cada iteração, a ordenação por inserção remove um elemento dos dados de entrada, localiza o local a que pertence na lista ordenada e o insere lá. Ele se repete até que nenhum elemento de entrada permaneça.

A ordenação geralmente é feita no local, iterando o vetor, aumentando a lista ordenada atrás dela. Em cada posição do vetor, ele verifica o valor lá em relação ao maior valor na lista ordenada (que acontece ao lado dele, na posição anterior da matriz marcada). Se maior, ele deixa o elemento no lugar e passa para o próximo. Se menor, ele encontra a posição correta na lista ordenada, muda todos os valores maiores para criar um espaço e insere na posição correta.

SELECT SORT

A ordenação de seleção é notada por sua simplicidade e possui vantagens de desempenho em relação a algoritmos mais complicados em determinadas situações, particularmente onde a memória auxiliar é limitada.

O algoritmo divide a lista de entrada em duas partes: a sub-lista de itens já ordenados, que é criada da esquerda para a direita na frente (esquerda) da lista e a sub-lista de itens restantes a serem ordenados que ocupam o restante do Lista. Inicialmente, a sub-lista ordenada está vazia e a sub-lista não ordenada é a lista de entrada inteira. O algoritmo prossegue localizando o elemento menor (ou maior, dependendo da ordenação) na sub-lista não ordenada, trocando pelo elemento não ordenado mais à esquerda (colocando-o sub-lista ordenada) e movendo os limites da sub-lista um elemento para a direita. A ordenação por seleção quando comparada a outras técnicas de classificação: a eficiência de tempo da ordenação por seleção é quadrática; portanto, existem várias técnicas de ordenação com melhor complexidade de tempo do que a Ordenação por Seleção.

SHELL SORT

Shellsort é uma generalização da ordenação por inserção que permite a troca de itens distantes. A idéia é organizar a lista de elementos para que, iniciando em qualquer lugar, considerando cada elemento h forneça uma lista ordenada. Equivalentemente, ele pode ser considerado como h listas intercaladas, cada uma classificada individualmente. Começando com grandes valores de h , esse rearranjo permite que os elementos movam longas distâncias na lista original, reduzindo a desordem rapidamente e deixando menos trabalho para as etapas menores de h - class. Seguindo essa ideia, uma sequência decrescente de valores de h que termina em 1 é garantida para deixar uma lista classificada no final.

QUICK SORT

MERGE SORT

The diagram illustrates the merging of 8 sorted arrays into a single sorted array. The arrays are represented as boxes containing numbers, and the merging steps are indicated by arrows with numbers 1 through 20.

Initial Arrays (Level 0):

- Array 1: 38, 27, 43, 3
- Array 2: 9, 82, 10


Merging Steps:

- Array 1 and Array 2 are merged into Array 3 (38, 27) and Array 4 (43, 3).
- Array 3 and Array 4 are merged into Array 5 (38, 27) and Array 6 (43, 3).
- Array 5 and Array 6 are merged into Array 7 (38, 27, 43, 3).
- Array 7 and Array 8 (9, 82, 10) are merged into Array 9 (9, 82) and Array 10 (10).
- Array 9 and Array 10 are merged into Array 11 (9, 82) and Array 12 (10).
- Array 11 and Array 12 are merged into Array 13 (9, 10, 82).
- Array 13 and Array 14 (3, 27, 38, 43) are merged into Array 15 (3, 9, 10, 27, 38, 43, 82).

Final Array (Level 4):

- Array 15: 3, 9, 10, 27, 38, 43, 82

1. Divida a lista não ordenada em n sub-listas, cada uma contendo um elemento (uma lista de um elemento é considerada ordenada).
2. Mesclar sublistas repetidamente para produzir novas sub-listas ordenadas até restar apenas uma lista, esta lista estará ordenada.



3	9	10	27	38	43	82
---	---	----	----	----	----	----

O Quicksort é executado recursivamente e, quando uma partição de tamanho menor ou igual a L , essa partição deverá ser imediatamente ordenada usando o algoritmo de Inserção

O Quicksort é executado recursivamente e, quando você obtiver uma partição de tamanho menor ou igual a L , essa partição deverá ser ignorada e continua particionando as partições maiores. Uma vez que todas as partições tenham comprimento menor ou igual a L , o algoritmo deverá executar o algoritmo de inserção.

Foi adicionado um parâmetro de controle para a chamada da função, para ao fim da primeira chamada na pilha de recursão executar o insert sort no vetor quase ordenado.

MERGE SORT COM INSERÇÃO PARCIAL

O Mergesort é executado recursivamente e uma vez que suas listas da esquerda e direita tiverem comprimento menor ou igual a L, elas deverão ser ordenadas independentemente usando o algoritmo de Inserção e, então, efetivamente mescladas.

MERGE SORT COM INSERÇÃO FINAL

Mergesort deverá ser executado recursivamente (como padrão) e uma vez que suas listas da esquerda e direita tiverem comprimento menor ou igual a L, o Mergesort deve para e imediatamente mesclar essas duas listas (certamente, isso não faria sentido desde o início). A mesclagem é feita com listas não ordenadas). Finalmente o algoritmo de Inserção deve ser executado em toda a lista.

Foi adicionado um parâmetro de controle para a chamada da função, para ao fim da primeira chamada na pilha de recursão executar o insert sort no vetor quase ordenado.

RESULTADOS

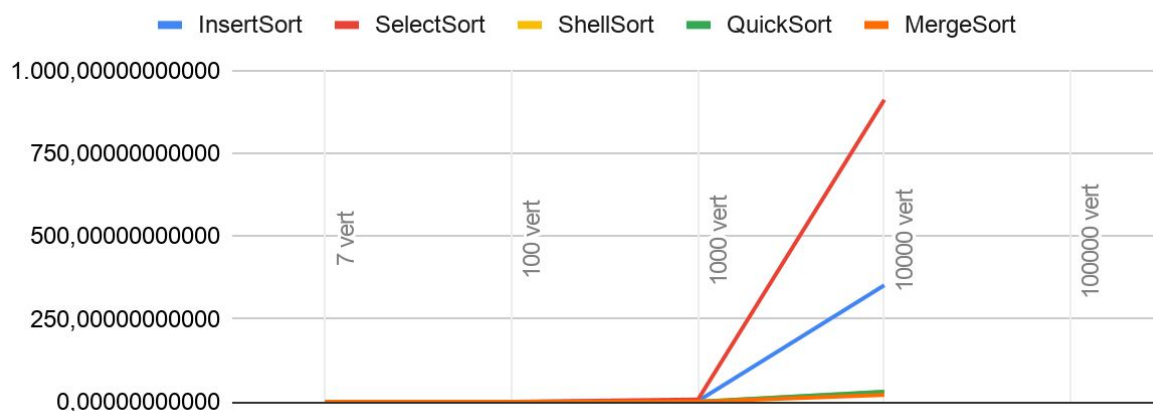
os algoritmos foram testados com arquivos de 7, 100, 1.000 e 10.000 vértices, o arquivo de 100.000 vértices estava demorando muito na minha máquina. Dentre os teste foram analisados quantidade de atribuições, comparações, e tempo de execução.

Os algoritmos Select Sort, Insert Sort, Shell Sort, Quick Sort, Merge Sort, foram testados e comparados juntos em uma tabela, Os algoritmos de Inserção parcial, e inserção final foram testados separadamente com o parâmetro L igual a 5 e a 20, e seus resultados foram comparados com o respectivo algoritmo sem a inserção parcial, ou final.

ALGORITMOS SEM INSERÇÃO FINAL OU PARCIAL.

Tempo (segundos)					
Algoritmos	7 vert	100 vert	1000 vert	10000 vert	100000 vert
InsertSort	0,00050401687	0,0172116	2,4581813	351,8666	---
SelectSort	0,00005338190	0,0301151	7,1043227	911,8337	---
ShellSort	0,00047898200	0,0042715	0,2303829	30,4257	---
QuickSort	0,00052237510	0,0058126	0,2668576	30,5455	---
MergeSort	0,00052571200	0,0049851	0,2331364	20,9502	---

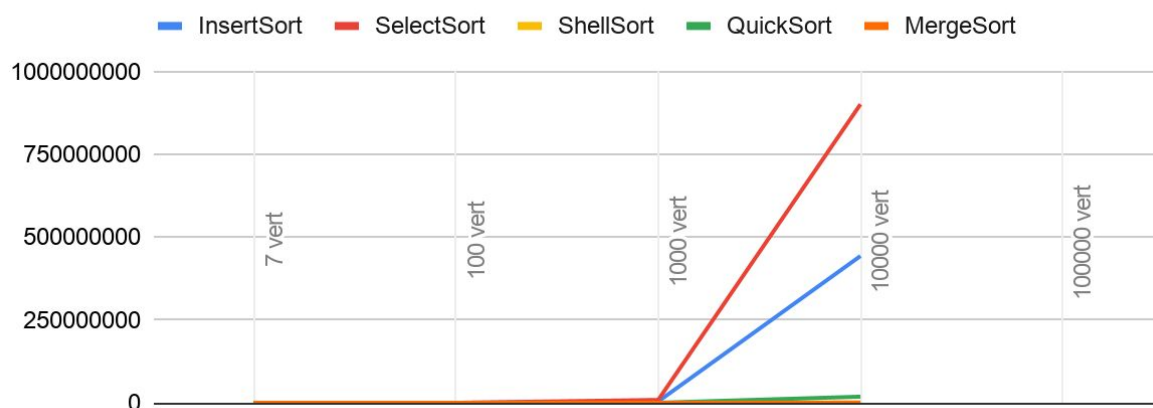
Tempo (segundos)



O Select sort obteve o menor tempo de execução no menor arquivo, porém teve o pior na maior arquivo testado, O Shell Sort Obteve o melhor tempo de execução nos arquivos intermediários, e no maior arquivo o melhor tempo de execução foi do Merge Sort.

Comparações					
Algoritmos	7 vert	100 vert	1000 vert	10000 vert	100000 vert
InsertSort	48	28059	4343395	443328863	---
SelectSort	122	59537	8820901	901440577	---
ShellSort	45	2744	65787	764790	---
QuickSort	103	8966	239764	18591728	---
MergeSort	84	3721	66775	877283	---

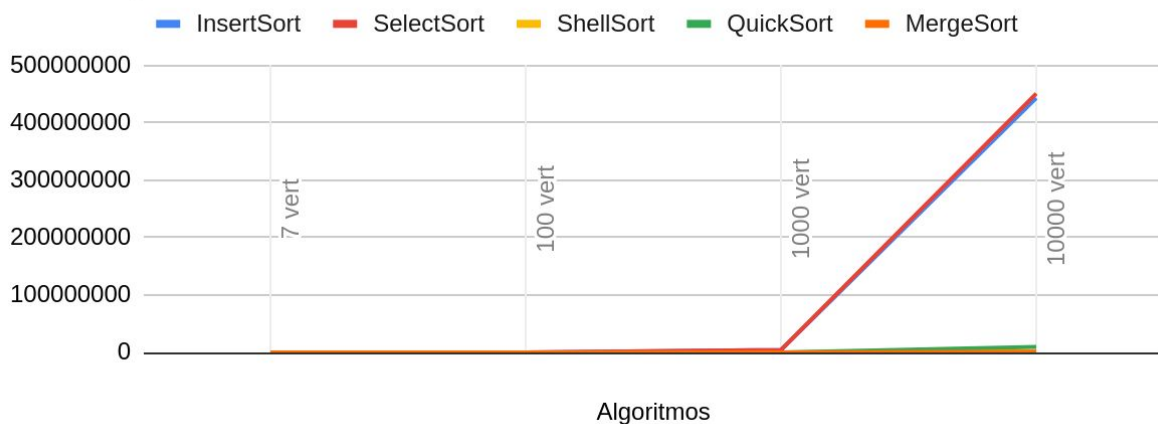
Comparações



No quesito número de comparações, o Shell Sort foi o melhor em todos os casos estudados, e o Select Sort foi o pior em todos.

Atribuições					
Algoritmos	7 vert	100 vert	1000 vert	10000 vert	100000 vert
InsertSort	81	28791	4352305	443418935	---
SelectSort	123	31325	4434003	450960235	---
ShellSort	140	7337	154003	1862161	---
QuickSort	118	5607	146325	9641584	---
MergeSort	157	6792	115418	1462940	---

Atribuições

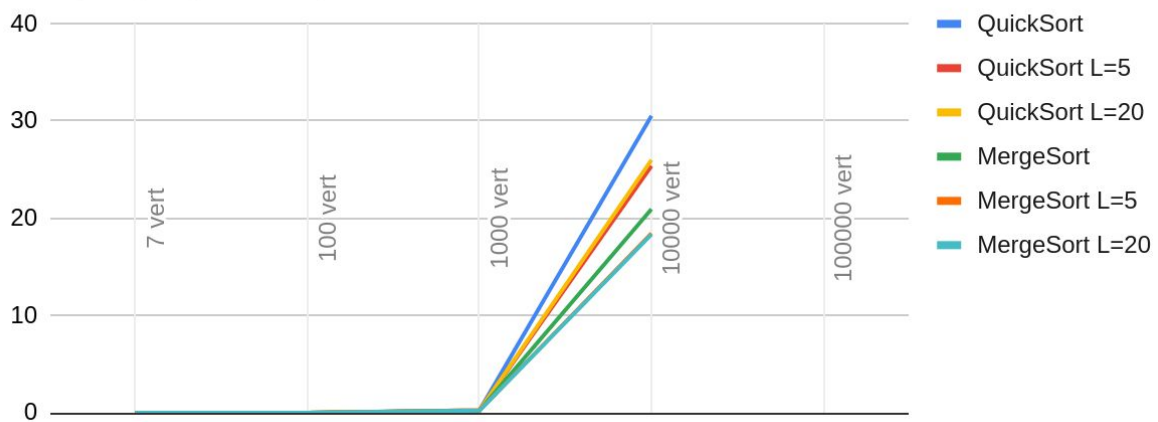


No menor arquivo o Insert Sort obteve o melhor resultado, no segundo o Quick Sort, e nos maiores arquivos o Merge Sort obteve os melhores resultados, e os piores foram o Select Sort e o Insert sort.

ALGORITMOS COM INSERÇÃO PARCIAL.

Tempo (segundos)					
Algoritmos	7 vert	100 vert	1000 vert	10000 vert	100000 vert
QuickSort	0,00052238	0,005812640	0,2668576	30,54550	---
QuickSort L=5	0,00047588	0,005724668	0,2667429	25,39489	---
QuickSort L=20	<i>0,00051141</i>	<i>0,004436016</i>	<i>0,2572067</i>	26,00179	---
MergeSort	0,00052571	0,004985094	0,2331364	20,95022	---
MergeSort L=5	0,00051570	0,004197359	0,2205505	18,45258	---
MergeSort L=20	0,00047278	0,004317522	0,2237408	18,35710	---

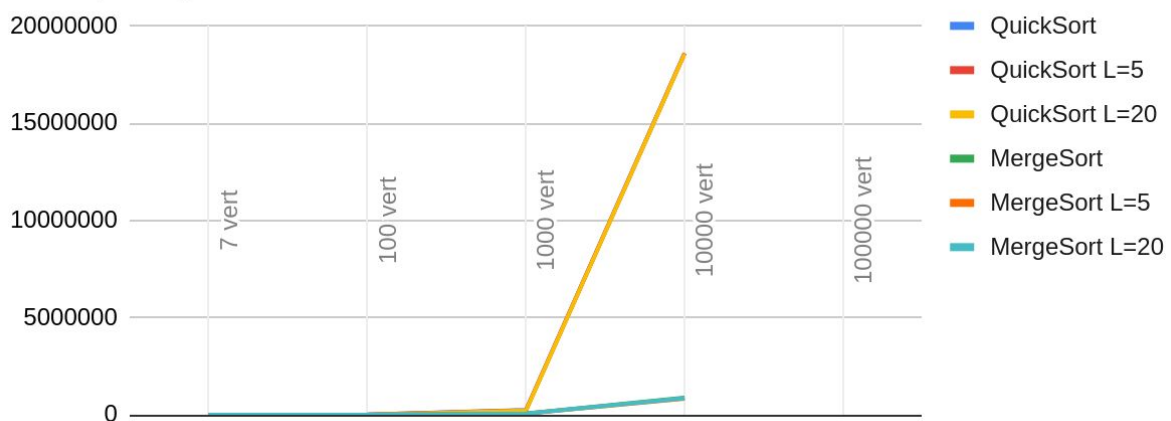
Tempo (segundos)



Em nenhum dos arquivos os algoritmos originais tem uma performance melhor que os com inserção parcial.

Comparações					
Algoritmos	7 vert	100 vert	1000 vert	10000 vert	100000 vert
QuickSort	103	8966	239764	18591728	---
QuickSort L=5	82	8724	238562	18590526	---
QuickSort L=20	60	4826	219062	18571026	---
MergeSort	84	3721	66775	877283	---
MergeSort L=5	70	3473	63633	829542	---
MergeSort L=20	51	3549	62125	856514	---

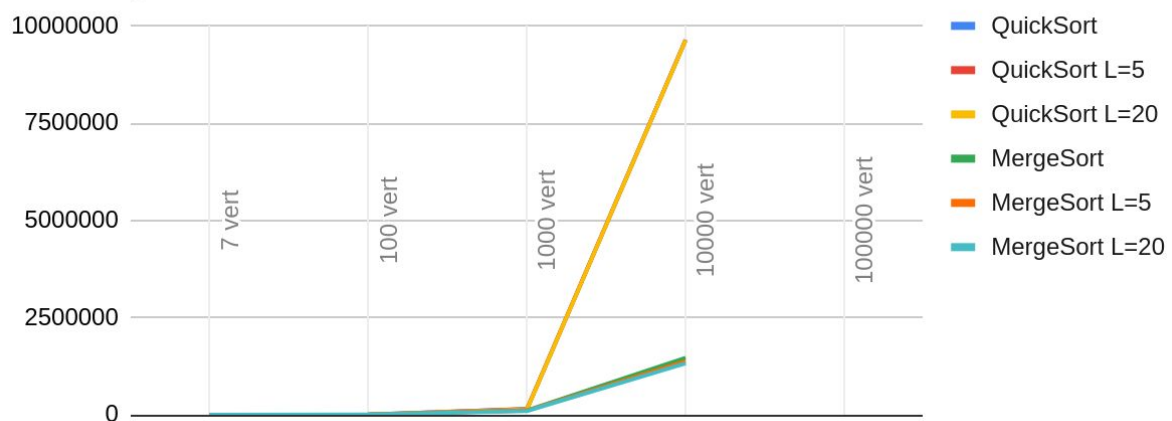
Comparações



Em nenhum dos arquivos os algoritmos originais tem uma performance melhor que os com inserção parcial.

Atribuições					
Algoritmos	7 vert	100 vert	1000 vert	10000 vert	100000 vert
QuickSort	118	5607	146325	9641584	---
QuickSort L=5	116	5577	146175	9641434	---
QuickSort L=20	105	4085	138675	9633934	---
MergeSort	157	6792	115418	1462940	---
MergeSort L=5	134	5964	108914	1372462	---
MergeSort L=20	101	5506	99932	1323262	---

Atribuições

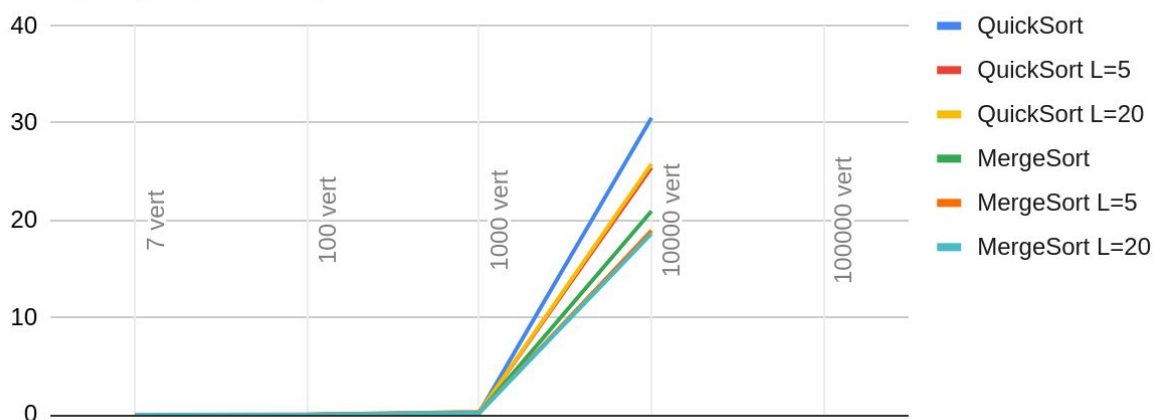


Em nenhum dos arquivos os algoritmos originais tem uma performance melhor que os com inserção parcial.

ALGORITMOS COM INSERÇÃO FINAL.

Tempo (segundos)					
Algoritmos	7 vert	100 vert	1000 vert	10000 vert	100000 vert
QuickSort	0,00052238	0,005812640	0,2668576	30,54550	---
QuickSort L=5	0,01006007	0,017193555	0,2924368	25,43231	---
QuickSort L=20	0,00047731	0,004544973	0,2748442	25,79883	---
MergeSort	0,00052571	0,004985094	0,2331364	20,95022	---
MergeSort L=5	0,00051689	0,003977770	0,2134075	18,94681	---
MergeSort L=20	0,00048089	0,005426407	0,2098846	18,62829	---

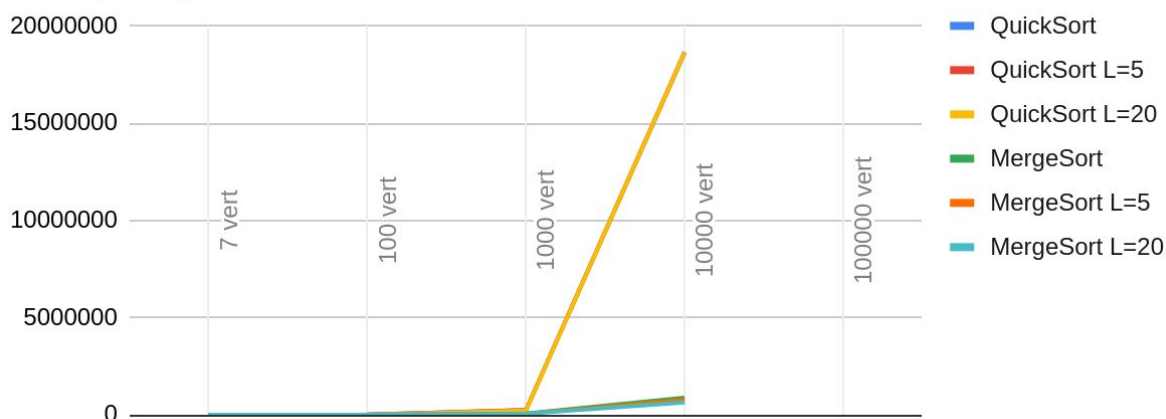
Tempo (segundos)



Para o menor arquivo o Quick Sort com inserção final e $L = 5$ foi ineficiente, sendo mais em arquivos maiores, o Merge Sort com inserção final na maioria dos casos foi mais eficiente, sendo que parâmetros L maiores obtêm melhores resultados em arquivos maiores.

Comparações					
Algoritmos	7 vert	100 vert	1000 vert	10000 vert	100000 vert
QuickSort	103	8966	239764	18591728	---
QuickSort L=5	98	9143	244153	18650225	---
QuickSort L=20	64	4853	223153	18629225	---
MergeSort	84	3721	66775	877283	---
MergeSort L=5	55	3072	57923	762386	---
MergeSort L=20	66	6194	54480	628385	---

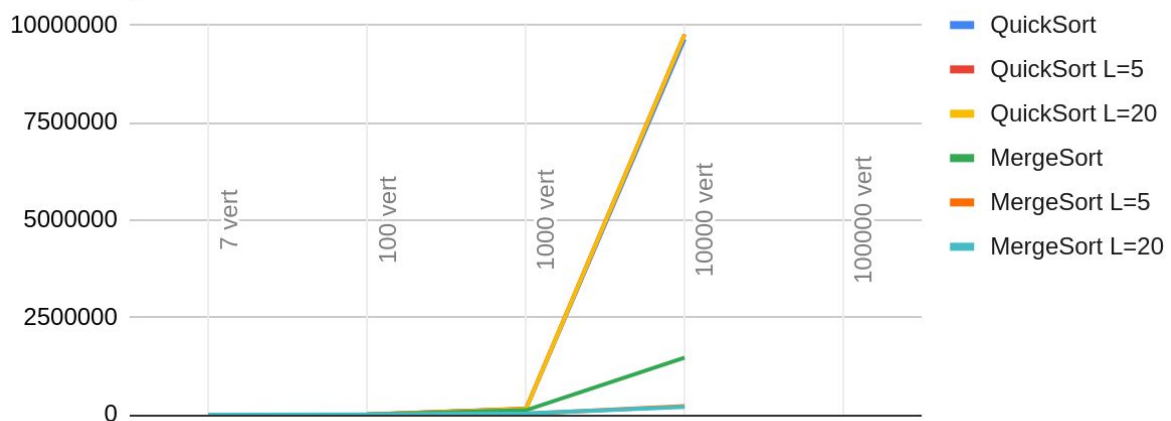
Comparações



A utilização de inserção parcial reduz o número de comparações, em relação aos algoritmos originais.

Atribuições					
Algoritmos	7 vert	100 vert	1000 vert	10000 vert	100000 vert
QuickSort	118	5607	146325	9641584	---
QuickSort L=5	149	6434	157456	9760931	---
QuickSort L=20	114	4289	146956	9750431	---
MergeSort	157	6792	115418	1462940	---
MergeSort L=5	64	1768	23213	220798	---
MergeSort L=20	91	5850	30098	199819	---

Atribuições



No caso do Merge Sort a inserção final reduziu drasticamente o número de atribuições em relação a implementação tradicional do mesmo, porém o Quick Sort não foi beneficiado neste aspecto, pelo contrário, o número de atribuições subiu.

DISCUTINDO A SUBSTITUIÇÃO DO ALGORITMO DE INSERÇÃO PELO SELECTION E PELO SHELL SORT

A substituição do algoritmo de inserção pelo Shell Sort ou pelo Select Sort pode ser feita desde que leve em consideração os seguintes aspectos:

Na inserção parcial, as partições são ordenadas imediatamente quando possuem tamanho menor que L, se o parâmetro L for pequeno não haverá ganho de performance se trocado pelo Select sort tendo em vista que possuem no pior caso a mesma complexidade, e a partição seria pequena. para L maiores, seria interessante trocar o insert Sort pelo Shell tendo em vista sua melhor performance em conjuntos maiores.

Na inserção final, o vetor é ordenado após ser particionado por inteiro, e estar “quase” ordenado, logo não seria interessante trocar o Insert Sort pelo Select, considerando que o conjunto em questão está próximo do melhor caso para o Insert, também não faz sentido trocar pelo shell tendo em vista que o shell reduz a desordem aos poucos para insert ordenar (shell com h igual a 1), o que já é feito pelos partições do Merge e o Quick