Trabalho Prático 01 de Estruturas de Dados – Escalonador de Urls

Juan Braga[1]

[1]Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – MG - Brasil

email (matrícula)

1. Introdução

A fim de entregar resultados de pesquisa úteis e atualizados, ferramentas de busca (como `www.google.com`, da Google LLC) têm o difícil trabalho de se manter a par da grande quantidade de páginas existentes na *internet*, o que envolve acessar e salvar o conteúdo de todas elas, atualizando-as periodicamente. Este processo, chamado de *web-crawling*, possui um grande custo computacional e de consumo de banda (como limites presentes tanto na velocidade de acesso da ferramenta quanto nos servidores em que os *sites* são hospedados), e certamente é sensível a otimizações de desempenho, em especial na determinação de **quais** páginas o *crawler* deve fazer a coleta/atualização. Uma vez que não é possível fazer a coleta simultânea de todas as páginas da *Web*, é preciso determinar a **ordem** com que visitar os *websites*, e buscar um equilíbrio entre a quantidade de páginas acessadas e a frequência com que elas são atualizadas: para esta tarefa incumbem-se os escalonadores (ou *schedulers*).

O princípio de funcionamento de um escalonador é o de que todas as entradas (neste contexto, os endereços *web*) que lhe são fornecidas serão devolvidas de maneira ordenada, de acordo com uma política de ordenação definida.

O trabalho documentado por este documento empregará a linguagem C++ na implementação um programa que escalona endereços de páginas web (também conhecidas como *URLs*) com o propósito de realizar uma análise da estrutura de dados **fila de listas** (escolhida pelo autor), ambas encadeadas e alocadas dinamicamente. A política de ordenação utilizada por este projeto seguirá uma estratégia *depth-first*, e se pautará pela ordem com que os *Hosts*[1] das páginas forem conhecidos, e os endereços a eles pertencentes serão ordenados seguindo o nível de profundidade do *link* (endereço).

Outras ações mais específicas também podem ser incumbidas ao escalonador.

---

[1] Um *Host*, no contexto de endereços web, é visto como o endereço de rede para acessar um computador ou servidor específico, como `website.com` é o *Host* da página `http://www.website.com/subpagina`. Este trabalho também irá se referir a este endereço como o "nome" do *Host*.

2. Método

2.1 Especificações

A primeira estrutura (Fila) representará cada um dos *Hosts*, e cada um destes (seus elementos) será uma outra estrutura (Lista) contendo os endereços *web* a eles pertencentes. A funcionalidade principal do programa, o escalonamento, se trata da escrita dos endereços em um arquivo de saída (*Urls* essas que deverão ser obtidas a partir de um arquivo de entrada), processo que os remove da estrutura, mas mantém o registro do *Host*.

Além de escalonar indiscriminadamente e adicionar endereços já de maneira ordenada, o escalonador pode ser solicitado a: escalonar uma quantidade específica de *Urls*, e/ou de algum *Host* também específico; exibir a lista de *Hosts* conhecidos; exibir as *Urls* de um *Host* específico; e limpar os endereços de *Hosts* ou de toda a estrutura.

A profundidade das *Urls* será definida pelo número de barras ('/') após o *Host* de cada endereço (desconsiderando-se a última não sucedida de subpáginas, como "http://www.ufmg.br/"), e apenas aquelas que seguirem o protocolo *HTTP* serão aceitas pelo escalonador. Arquivos com as extensões .jpg, .gif, .mp3, .avi, .doc e .pdf, deverão ser ignorados, assim como fragmentos após o símbolo #. Por fim, a presença de "www." antes do *Host* deverá ser desconsiderada e removida pelo escalonador.

Assim, como exemplo da ordenação, se uma página pertencente ao *Host* `umapagina.com` for fornecida ao escalonador antes de uma outra que pertença ao *Host* `outrapagina.com`, todas as *Urls* de `umapagina.com` serão escalonadas antes de qualquer endereço de `outrapagina.com`; e o endereço `http://www.umapagina.com/subpagina` (de nível 1) sempre terá prioridade em relação a `http://www.umapagina.com/subpagina/subsubpagina` (de nível 2). Em casos de nível semelhante, as *Urls* recebidas primeiro terão prioridade.

2.2 Estrutura de dados

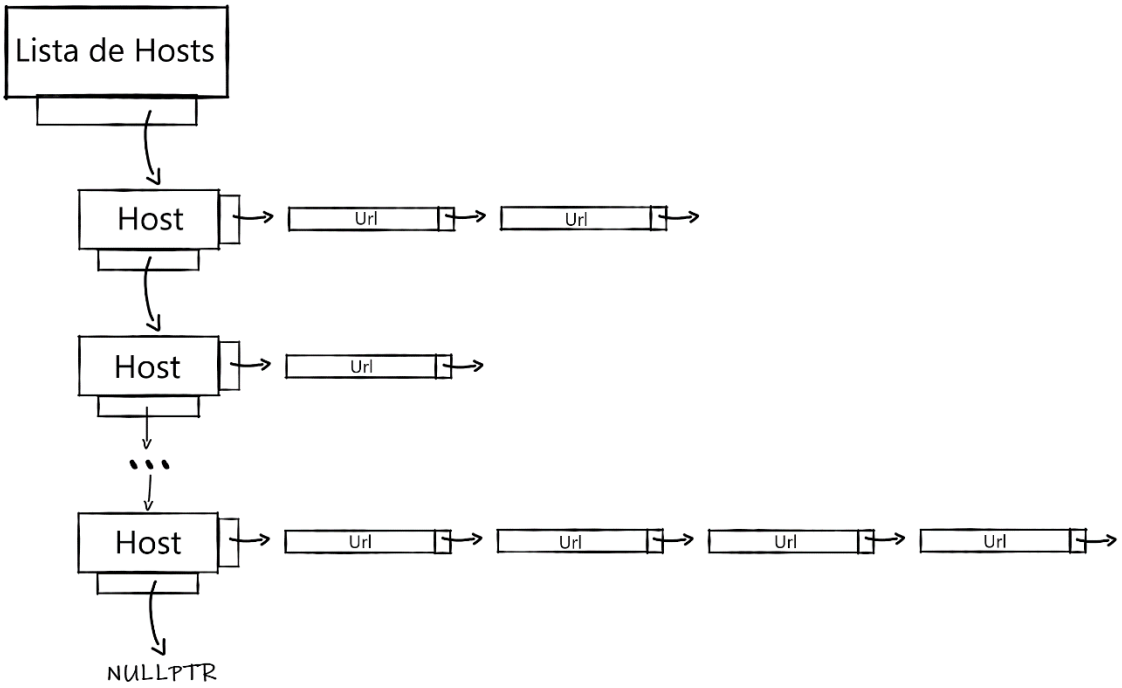
A estrutura empregada para o programa escalonador foi uma fila encadeada de listas encadeadas, onde a primeira contém cada um dos *Hosts* como seus nós, na ordem em que o escalonador os conheceu, e a segunda (ou seja, cada um de seus nós) constitui outra lista que deverá armazenar os endereços (*Urls*) ordenados de acordo com sua profundidade. É preciso salientar, contudo, que a complexidade da fila diferirá do esperado (de uma estrutura comum do tipo "fila"), dada a necessidade de percorre-la fila ao inserir endereços - o que permite que ela possa ser vista como apenas mais uma lista encadeada.

Definiu-se a adoção da estrutura em questão por conta de o tamanho da lista de *Hosts* não ser conhecido de antemão[2], e, portanto, exigir uma flexibilidade de memória que o encadeamento dinâmico proporciona. A conjunção de listas secundárias para cada *host* da fila, por sua vez, se deu em razão de reduzir o custo de busca dos e inserção nos *Hosts* (que comparado a uma lista única, com todos os endereços ordenados juntos, não requere verificar todos os endereços de cada *Host* até encontrar aquele desejado).

Por fim, o autor reconhece que a utilização da própria lista para manter o registro dos *Hosts* conhecidos pode introduzir um custo adicional na procura de elementos, e é possível que a utilização de uma fila adicional apenas para armazenamento dos *Hosts*, em conjunto com a funcionalidade de remover o *Host* caso ele esteja vazio, seria uma solução alternativa com um aumento de performance em casos com grande quantidade de *Hosts*.

Uma representação da estrutura pode ser conferida no diagrama da Figura 1:

Figura 1 - Representação de lista encadeada de listas encadeadas



[2] Para as finalidades do escalonador, o autor deve reconhecer que uma estrutura como tabela *hash* aliada com hardware dedicado provavelmente seria uma escolha mais eficiente, uma vez que o acesso aos *Hosts* teria a complexidade assintótica significamente melhorada ao mapeá-los - em comparação com uma lista encadeada em que seja preciso iterar por um número de elementos para se encontrar um *Host* desejado. Contudo, pela falta de familiaridade prática com tal estrutura por parte do autor, uma estrutura mais simples foi escolhida.

2.3 Classes

As classes do projeto foram planejadas antes da implementação, e o autor optou por uma estrutura de dados especificamente personalizada para a situação em questão. Isto significa que os métodos presentes foram desenhados do zero com o programa em mente, em comparação com a reutilização e adaptação de estruturas mais genéricas previamente implementadas pelo autor.

Reconhecidamente, o *mantra* de abstração e reutilização de código foi levemente violado ao adotar esta estratégia mais "artesanal", e é preciso conceder que esforços poderiam talvez ter sido salvos caso estruturas mais genéricas tivessem sido criadas, com métodos virtuais, e modificadas para atentar às especificações do trabalho. Um exemplo seria a reutilização de uma lista encadeada para formular tanto a "fila" de *Hosts* quanto a "lista" de *Urls*.

Um aspecto adicional de modularização do código poderia ter recebido mais atenção: o acesso a arquivos. A estrutura em seu estado final se faz dependente de variáveis globais para a escrita dos resultados em arquivo pelas funções de "impressão" ou "visualização", uma vez que certos comandos requerem que várias *Urls* precisem ser processadas e impossibilitam a passagem entre as funções.

2.3.1 ListaHosts

Armazena uma lista encadeada que armazena os *Hosts*, e tem o papel de receber *Urls* (em formato string) e inseri-las neles, criando *Hosts* novos no fim da fila caso não tenham sido conhecidos anteriormente. A validade da *Url* é averiguada, mas o real custo da verificação foi delegado à classe `Url` (ver seção 2.3.3).

Além do mais, a estrutura possui a responsabilidade de fornecer a maior parte das funcionalidades esperadas do escalonador[3], como as de:

1. Escalonamento: `escalona(quantidade)` que percorre os *Hosts* na ordem em que foram conhecidos e escalona as *Urls* de cada um (exibe[4] e remove da estrutura) até que não haja mais elementos ou a quantidade solicitada tenha sido fornecida, `escalona(host,quantidade)` que busca o *Host* desejado e escalona a quantidade especificada, e `escalonaTudo()` que escalona todas as *Urls* de todos os *Hosts*;

[3] Neste contexto há uma consideração a ser feita: chamar a estrutura de `ListaHosts` e considerar o programa como `Escalonador`, ou já nomear a classe como o próprio escalonador, onde a função `main` seria vista apenas como uma auxiliar para manipular os arquivos e parâmetros. O autor optou pela primeira opção.

[4] No âmbito da implementação deste trabalho, o termo "exibir" deve ser compreendido como a impressão de conteúdo em arquivo, a menos que explicitamente especificado.

2. Exibição: `verHost(host)` que exibe as *Urls* pertencentes a um *Host* (sem remover) e `listaHosts()` que exibe uma lista com todos os *Hosts* que já foram vistos pelo escalonador (mesmo que tenham mais endereços); e
3. Limpeza: `limpa(host)`, responsável por esvaziar um dado *Host*, e `limpaTudo()` que esvazia todos os *Hosts* e a lista de *Hosts* também.

Os métodos auxiliares `procuraHost(host)`, `imprime()` e `acessa()` foram definidos para facilitar algumas funcionalidades externas, respectivamente encontrar o endereço de um *Host* especificado, imprimir no terminal (em contraste à escrita em arquivo do escalonamento) e acessar todos os elementos da estrutura para propósitos de análise experimental (ver seção 5), e foram mantidos públicos.

Adicionalmente, foram disponibilizados os métodos `escalonaTudoBreadhFirst()`, `escalonaBreadthFirst(quantidade)`, `escalonaTudoBestFirst()` e `escalonaBestFirst(quantidade)`, para o cumprimento dos desafios.

2.3.2 Host

A classe `Host` foi pensada para uso exclusivo dos métodos da classe `ListaHosts`, porém se encontra exposta no cabeçalho `escalonador.h` e deve ser usada com cautela. Sua responsabilidade, além de se comportar como nó para a classe `ListaHosts` é receber endereços dela e inseri-los de maneira que aqueles com maior profundidade fiquem ao fim da lista. Como afirmado anteriormente, o critério de desempate para endereços de mesmo nível foi definido como a ordem em que eles chegam ao conhecimento do escalonador (ver seção 2.1). Esta classe também tem o dever de escalonar, exibir e remover endereços quando chamada pelos métodos da classe `ListaHosts`.

Os utilitários providos à `ListaHosts` por esta classe são, além da inserção, os de: escalonar uma *Url* de um *Host* (`escalona()`), exibir os endereços de um *Host* (`verEnderecos()`), exibir o nome do *Host* (`verHost()`) e limpar todas as suas *Urls* (`esvazia()`). Métodos *getters* para recuperar o nome (`nome()`), tamanho (`tamanho()`) ou o próximo *Host* apontado (`proximo()`) também foram implementados, assim como um para checar se a lista não contém elementos (`vazio()`).

Para fins de simplicidade, apenas um método `escalona()` foi criado para escalonar apenas um endereço do *Host* que o chamar, para fins de simplicidade. Contudo, seria também possível a criação de um método `escalona(quantidade)` que poderia fornecer alguma economia em comparações e tempo de execução.

Para a análise do uso de memória, dois métodos foram adicionados: para imprimir os endereços no terminal (`imprimeEnderecos()`) e acessar os endereços (`acessaEnderecos()`).

2.3.3 Url

Tem o papel simples de armazenar um endereço e se comportar como um nó da classe `Host`.

Foi definido que na instanciação da classe se realize o processamento da *Url* recebida para se determinar a validade e nível, informações que também deverão ser armazenadas dentro da classe para recuperação conveniente através de métodos.

Os métodos disponíveis na classe são `eValida()`[5], que retorna o dado de se o endereço contido segue as especificações (ver seção 2.1), `extraiHost()`, que retorna apenas o nome do *Host* do endereço, `imprime()`, que armazena no arquivo de saída o endereço da *Url* que o chama, e `nivel()`, que retorna o nivel da *Url* se for válida (ou -1 se não o for). Ademais, o método `processa()` foi criado para uso interno e será executado na instanciação para determinar a validade e profundidade do endereço.

2.4 Main

O arquivo `main.cpp` serve como auxiliar para a seleção de comandos e leitura/escrita dos dados, através do processamento de parâmetros e do recebimento do arquivo através da chamada do programa. Adicionalmente, também define as fases e controla o uso da análise de acessos da biblioteca "memlog.h".

As funções definidas nela são: `menu(void)`, para exibir um menu de opções caso parâmetros inseridos na chamada do programa não sejam reconhecidos; `parseArguments(argc,argv)`, para auxiliar no processamento dos parâmetros e do nome do arquivo; `nomeSaida(nomeEntrada)`, para auxiliar na criação do nome do arquivo de saída dentro das especificações; `terminaCom(nomeEntrada,termino)`, para auxiliar na identificação para remoção da extensão ".txt" do nome do arquivo de entrada[6]; e `pegaComando(entrada)`, que captura uma linha do arquivo de entrada e o interpreta como um dos comandos ofertados pela aplicação.

3. Análise de Complexidade

3.1 Complexidade de tempo

Inserir: A inserção de uma *Url* válida na estrutura será pautada por três fatores, o processamento do endereço, a extração do nome do *Host* dele e o percorrimento pela estrutura. No processo de construção, dado um endereço de comprimento *n*, se tomará o número de operações como um múltiplo pequeno de *n*, resultando em *O(n)* e com peso

---

[5] É preciso notar que o artigo pode resultar em um prejuízo na semântica do código na análise da implementação, uma vez que os termos "Url" e "endereço" podem ser usados intercambiavelmente. O artigo feminino (*Valid**a***) foi escolhido por combinar com o artigo do nome da classe, *Url*.

[6] Adaptada de "Find out if string ends with another string in C++", Stackoverflow.

maior da operação de expressão regular, ou em praticamente *O(1)* em caso de uma *Url* de tamanho mínimo. A extração do nome terá complexidade *O(n)*, e novamente *O(1)* para um endereço pequeno.

No melhor caso, a inserção ocorrerá a primeira posição da `ListaHosts` e no início da "sublista", resultando em complexidade de percorrimento *O(1)*. No pior caso, *N Hosts* deverão ser percorridos até o último elemento, e *M* endereços de nível menor ou igual[7] percorridos até a posição da *Url*. Temos assim uma complexidade de percorrimento *O(N+M)*.

Assim, temos *O(1)* para o melhor caso, e *O(n) + O(n) + O(N+M) = O(max(n,max(N+M))*.

Apesar de não houver um limite no tamanho de *Urls*, é provável de a maior parte delas não exceder algumas centenas de caracteres, enquanto é certo que um escalonador poderá ter um número bem maior de elementos que o comprimento do endereço. Em um caso médio é razoável aferir que *max(n,N+M) = (N+M)*, e que a complexidade da inserção seja *O(N+M)*.

Escalonar: O escalonamento terá complexidade *O(1)*, pois retira apenas o primeiro elemento do *Host* fornecido.

Escalonar Host: O escalonamento de um Host específico envolverá o percorrimento de *N* elementos na lista[8]. A complexidade será de apenas *O(N)*.

Escalonar Tudo: O escalonamento total irá percorrer uma lista de *N Hosts* e imprimir todos os *M* endereços neles contidos, resultando em complexidade *O(NM)*.

Ver Host: A complexidade da visualização de um *Host* específico será influenciada pela busca pela lista de *Hosts* de tamanho *N*, e pela impressão de todos os *M* endereços dela, resultando em *O(N+M)*.

ListaHosts: A listagem irá imprimir os *N Hosts* contidos na lista, logo *O(N)*.

Limpar Host: Na limpeza o *Host* é buscado entre os *N* elementos da lista, e cada um dos *M* endereços são desalocados. Temos *O(N+M)*.

Limpar Tudo: Na limpeza total, todos os *N Hosts* são acessados, e seus *M* endereços são desalocados. Resultado: *O(NM)*

---

[7] O caso de todas serem iguais envolverá também a comparação de seus endereços, que poderia adicionar uma nova variável a se multiplicar por *M*.

[8] Cada "visita" ao elemento envolve a comparação do nome fornecido com strings de tamanhos variáveis, que poderiam ser consideradas como uma variável adicional na análise. Para os propósitos deste trabalho elas serão vistas apenas com complexidade *O(1)*.

3.2 Complexidade de espaço

A implementação da estrutura não permite que se insiram *Hosts* ou *Urls* repetidas, então a complexidade de espaço da estrutura de dados escolhida será simplesmente o número total de elementos que foram fornecidos ao escalonador, *M Urls*, somado de *N Hosts* extraídos a partir deles. Em termos gerais, a complexidade dela pode ser vista como *O(M+N)*, com *M* o número de entradas únicas e *N* o número de *Hosts* adicionados, e um pior caso ocorreria quando *N* atingisse seu valor máximo e igual a *M*, resultando em *O(2M)*.

Dado que a funcionalidade do programa é representar e organizar os dados recebidos, onde a única ocasião discrepante seria a não adição de *Url* repetida ou inválida, a complexidade de espaço se encontra dentro do esperado, com o número de elementos válidos sendo igual às *N* entradas recebidas, com a adição dos *M Hosts* extraídos a partir delas.

Dado que as demais funções não adicionam nenhum dado à estrutura, elas podem ser vistas com complexidade de espaço simplificada de *O(1)*, incluindo a inserção de *Url* válida com *Host* desconhecido (que criará uma instância de *Host* e uma *string* com o nome a mais, porém não escapa da complexidade definida anteriormente).

4. Estratégias de Robustez

A função `parseArguments(argc,argv)` deve ser capaz de lidar com parâmetros incorretos passados para o programa, assim como identificar quando um nome do arquivo de entrada não for fornecido.

A prioridade definida para o programa é a de que o processamento não deve ser interrompido por conta de uma entrada mal formatada. Em outras palavras, comandos incorretos, quantidades incorretas (de visão ou escalonamento), linhas de endereço insuficientes ou passadas como comando resultarão no programa ignorando a entrada. Muito provavelmente um erro na escrita de um comando de inserção incorrerá na leitura dos endereços seguintes como comandos incorretos, e quantidades incorretas levarão o programa a ler comandos como *Urls* e posteriormente descartá-los. Isto é comportamento esperado.

A biblioteca `msgassert.h` foi empregada para a utilização de funções de asserção (`erroAssert(condição,"mensagem-erro")`) nos locais onde não deveriam ocorrer erros durante as operações internas da estrutura, como alocações não realizadas, recebimento de parâmetros sem dados em funções internas (como `criaUrl()`, que se receber um endereço vazio significa que não há mais o que ser lido), erros no cálculo do tamanho das estruturas (variável `tamanho` conter número positivo mas o conteúdo for `nullptr`), criação de *Host* com *Url* inválida (ver parágrafo seguinte e seção 2.3.2), e se a inicialização do arquivo não tiver sido realizada na `main()`.

Adicionalmente, os métodos da estrutura `Host`, mais especificamente o construtor e o de inserção, receberam uso mais liberal de tal função (`erroAssert()`) em decorrência de ela ter sido planejada para uso por parte da `ListaHosts`. Uma readaptação para uso geral requereria apenas a substituição das asserções por condições que tratem dos casos.

5. Análise Experimental

A estrutura é formada por quatro entidades dinâmicas, `ListaHosts -> Host -> Url -> nome`. Para os propósitos desta análise será o local de escrita do conteúdo da `Url`, sua string, o ponto de foco. Também será contemplada uma análise da classe `Host` através do acesso a seu nome.

Os comandos para a verificação de acesso a ambas as classes também foram inseridos no código, mas serão comentadas para os propósitos da análise do desempenho seguindo os parâmetros definidos no parágrafo anterior.

Foram também definidas as seguintes fases: Inserção das Urls; Escalonamento de todos os endereços da estrutura; e uma terceira para a impressão da estrutura como um todo no terminal, sem o tempo de escrita no arquivo[9]. Vale lembrar que a fase definida para ambos o escalonamento e impressão foram definidas como "1" no arquivo da função `main()`, uma vez que o escalonamento removeria todos os elementos e impediria a impressão dos objetos.

O parâmetro de depuração "-g" foi removido do arquivo `Makefile`.

5.1 Considerações

Como não se usou um vetor tradicional de caracteres, se faz necessária uma breve análise do tipo `string` para averiguar sua semelhança com a outra estrutura. Buscando sua declaração, o tipo `typedef basic_string<char> string` (em `stringfwd.h`) é declarado, porém não é de grande ajuda. A implementação da classe `basic_string` (em `basic_string.h`) é feita em sua maior parte como um código *template*, e suas definições se encontram além da atual compreensão do autor.

Como "o custo de anexar caracteres a uma string se aproxima de uma constante assintoticamente conforme a string cresce"[10], o desempenho do tipo string será considerado equivalente ao de um vetor de caracteres na linguagem C. Por outro lado, as operações realizadas para processar o endereço contido na Url podem alterar a composição da estrutura alocada em relação à antes do processamento, o que oferece o

---

[9] Tal estratégia resultou em certa redundância em relação aos métodos similares para impressão no arquivo. Como a utilização de métodos que pudessem distinguir os parâmetros poderia gerar uma semântica menos intuitiva, optou-se por manter a redundância, mas reconhecer que tais funções não precisariam existir em um programa de escalonador comum.

[10] Guntheroth, K.

potencial de prejudicar uma análise da localidade de referência. Alternativamente, seria possível "dizer ao objeto string qual deve ser o tamanho do buffer original, então a menos que [seja excedido], ele não alocará blocos dinâmicos da memória"[11]. No contexto em que se tenha uma implementação pronta à base da estrutura `string` da *Standart Library*, foi decidido manter o seu uso e considerar o acesso ao endereço do primeiro caractere, fornecido pelo método `std::string::front()`, como o endereço de um vetor verdadeiro de caracteres, e desconsiderar as possíveis realocações internas.

Múltiplos acessos a uma instância de uma entidade dentro de uma operação serão contemplados apenas por uma utilização da biblioteca "memlog". Por exemplo, **na construção da entidade "Url" será registrado apenas um acesso à `string` com o endereço**, mesmo que múltiplas operações sejam feitas sobre ela - efetivamente, será presumido que o acesso pelo método foi capaz de carregar toda a entidade na memória principal.

De maneira semelhante, o uso do método `acessa()`, para carregar em memória os elementos, irá apenas fazer o acesso ao primeiro elemento, mas se concede que uma função auxiliar que percorra toda a string poderia também ser utilizada.

5.2 Especificação dos testes

Será feita uma bateria de cinco testes consecutivos para cada caso, aproximadamente dois minutos após o sistema operacional (Ubuntu 21.10 da Canonical foi utilizado neste trabalho) ter sido inicializado, sem a abertura de nenhum programa exceto o *Terminal*, para inserir o comando de execução, o programa "top", para terminal, a fim de checar a carga da máquina (mas encerrado antes da execução dos testes), e o programa "Files" do projeto Nautilus para a navegação nas pastas e manuseio dos arquivos de resultado. Foi utilizada uma máquina com processador *Intel® Core™ i5-4690K CPU @ 3.50GHz × 4*, sem *overclock*, e com uma memória RAM de 8GB, dividida em dois pentes DDR3 a uma frequência de 2133Mhz.

Os casos de teste serão produzidos usando o programa `geracarga` providenciado pelos professores da matéria, e seguirão o formato:

```
ADD_URLS <quantidade>
<elementos>
LISTA_HOSTS
ESCALONA_TUDO
LIMPA_TUDO
```

Casos: Inserção de elementos 10 Hosts x 1 Url, 10x5, 10x10, 100x10, 50x50, 100x50 e 100x100, com os respectivos comandos de criação:

[11] "How is std::string implemented?", Stackoverflow.

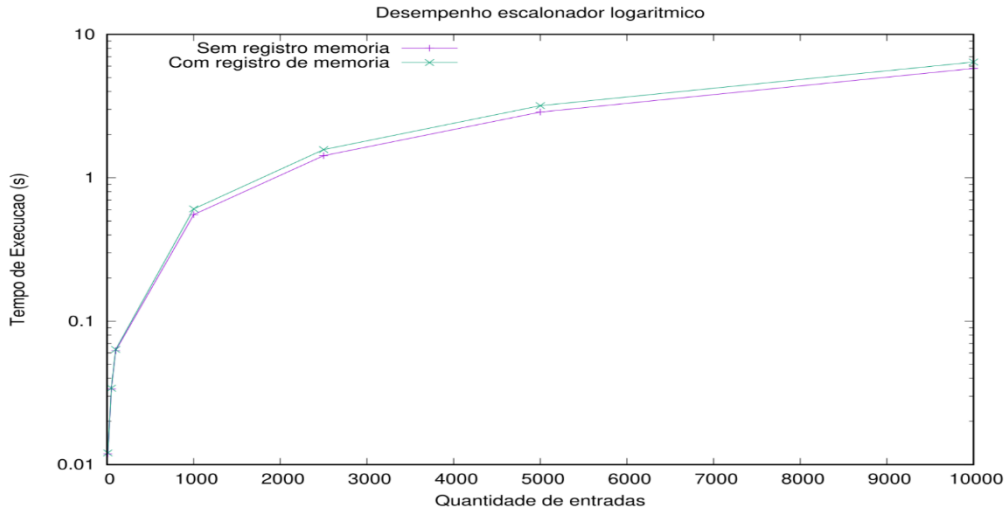
```
geracarga -s 10 -u 1 -p 5 -q 4 -v 4 -o carga10.txt
geracarga -s 10 -u 5 -p 5 -q 4 -v 4 -o carga50.txt
geracarga -s 10 -u 10 -p 5 -q 4 -v 4 -o carga100.txt
geracarga -s 100 -u 10 -p 5 -q 4 -v 4 -o carga1000.txt
geracarga -s 100 -u 50 -p 5 -q 4 -v 4 -o carga5000.txt
geracarga -s 50 -u 50 -p 5 -q 4 -v 4 -o carga2500.txt
geracarga -s 100 -u 100 -p 5 -q 4 -v 4 -o carga10000.txt
```

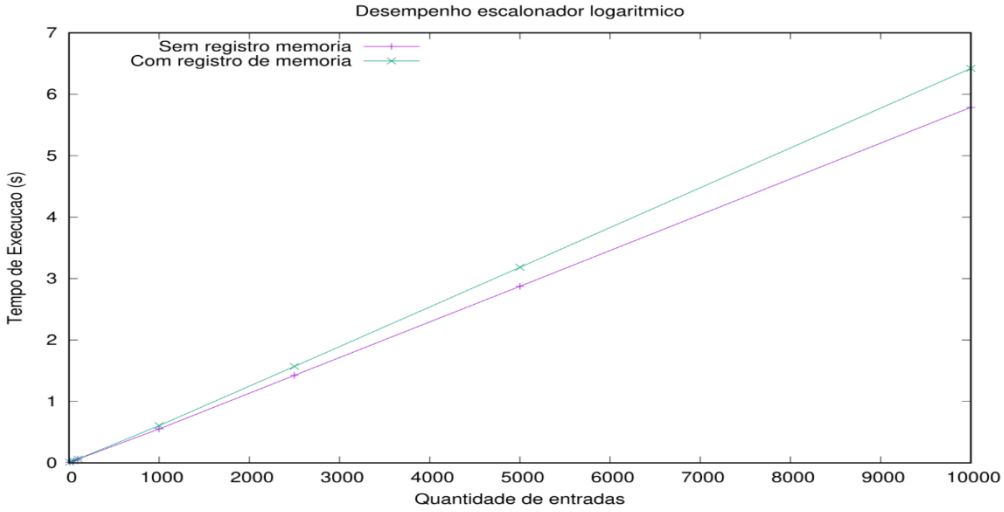
5.3 Performance

5.3.1 Desempenho

Será avaliada a performance de dois casos, com e sem o registro de memória:

Figura 2 - Gráfico de desempenho do algoritmo



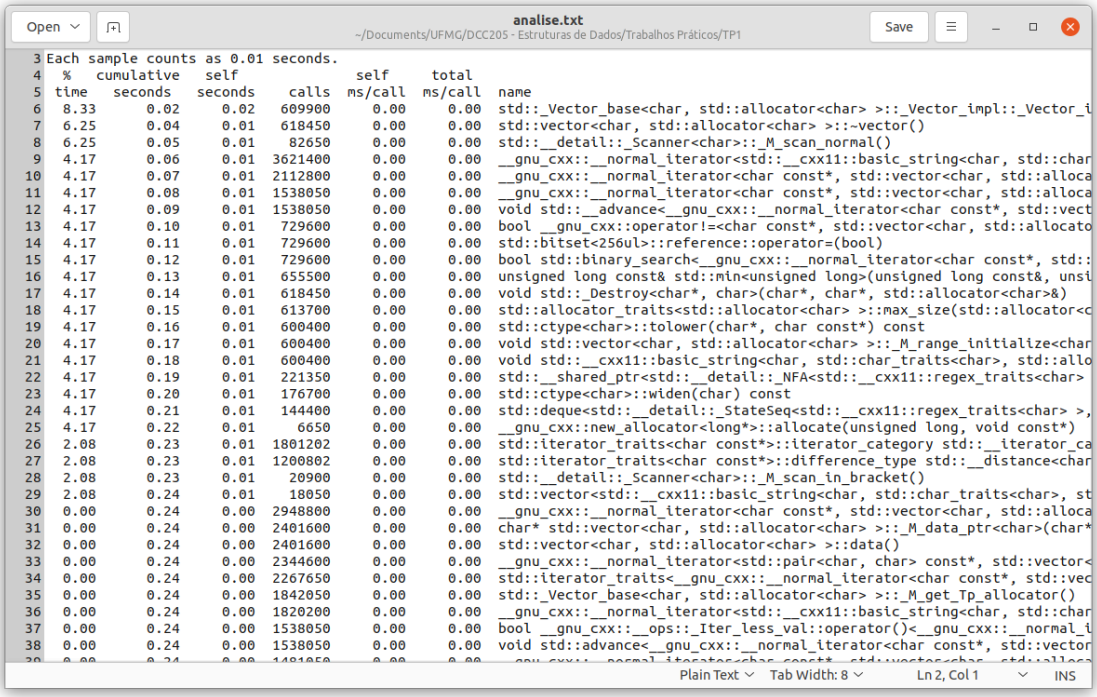


Percebe-se pelo gráfico que há uma progressão relativamente linear do tempo de processamento com a quantidade de operações realizadas, assim como um impacto significativo dos registros de memória - que apesar da diferença aparente pequena, em casos de entradas maiores certamente haveria um peso maior sobre a performance.

5.3.2 Depuração de desempenho

Com o objetivo de exacerbar as discrepâncias das chamadas de funções na depuração de desempenho, a carga de tamanho 10000 (ver seção 5.2) foi escolhida. Porém, foi notado que as funções de maior impacto foram as referentes a operações com o tipo string, em especial as relacionadas com a expressão regular, iteradores pelas strings, alocadores de tipo vector<> (de alguma funcionalidade interna do programa), entre outras (os resultados são infelizmente demasiado longos, e não poderão ser vistos em sua integridade neste trabalho, mas serão parcialmente mostrados na Figura 3).

Figura 3 – Porcentagem de uso de métodos e funções

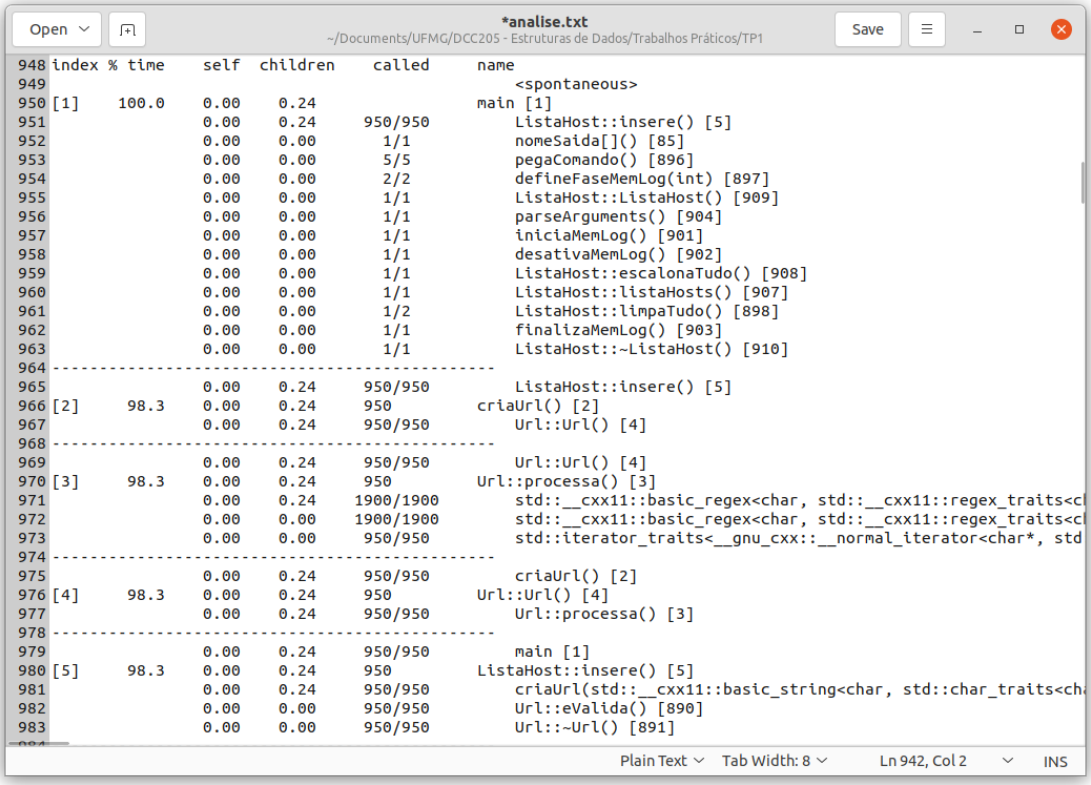


Desta forma, decidiu-se criar uma carga que permita uma menor variação e custo de processamento das strings, através do comando de criação `geracarga -o cargaespecial.txt -s 100 -u 10 -p 10 -q 5 -v 2`, do programa "geracarga" fornecido pelos professores.

Os resultados mostram que de fato o maior custo foi tomado pelo construtor da entidade `Url`, e pelas operações sobre as *strings*, especificamente o uso da função de comparação usando expressão regular e de iteradores, como pode ser visto no grafo da Figura 4, gerado a partir do programa `gprof` (GNU GCC Profiling Tool). Verifica-se,

portanto, que apesar de a complexidade da estrutura em si ter sido avaliada, não há escapatória do real peso de todas as operações com strings, cujas comparações e verificações, apesar de passiveis de melhorias, são inevitáveis.

Figura 4 - Grafo de Execução do Programa Escalonador




```
index % time    self  children    called     name
                                             <spontaneous>
[1]    100.0    0.00    0.24                 main [1]
                0.00    0.24     950/950         ListaHost::insere() [5]
                0.00    0.00       1/1           nomeSaida[]() [85]
                0.00    0.00       5/5           pegaComando() [896]
                0.00    0.00       2/2           defineFaseMemLog(int) [897]
                0.00    0.00       1/1           ListaHost::ListaHost() [909]
                0.00    0.00       1/1           parseArguments() [904]
                0.00    0.00       1/1           iniciaMemLog() [901]
                0.00    0.00       1/1           desativaMemLog() [902]
                0.00    0.00       1/1           ListaHost::escalonaTudo() [908]
                0.00    0.00       1/1           ListaHost::listaHosts() [907]
                0.00    0.00       1/2           ListaHost::limpaTudo() [898]
                0.00    0.00       1/1           finalizaMemLog() [903]
                0.00    0.00       1/1           ListaHost::~ListaHost() [910]
-----------------------------------------------
                0.00    0.24     950/950         ListaHost::insere() [5]
[2]     98.3    0.00    0.24     950           criaUrl() [2]
                0.00    0.24     950/950         Url::Url() [4]
-----------------------------------------------
                0.00    0.24     950/950         Url::Url() [4]
[3]     98.3    0.00    0.24     950           Url::processa() [3]
                0.00    0.24    1900/1900        std::__cxx11::basic_regex<char, std::__cxx11::regex_traits<cl
                0.00    0.00    1900/1900        std::__cxx11::basic_regex<char, std::__cxx11::regex_traits<cl
                0.00    0.00     950/950         std::iterator_traits<__gnu_cxx::__normal_iterator<char*, std
-----------------------------------------------
                0.00    0.24     950/950         criaUrl() [2]
[4]     98.3    0.00    0.24     950           Url::Url() [4]
                0.00    0.24     950/950         Url::processa() [3]
-----------------------------------------------
                0.00    0.24     950/950         main [1]
[5]     98.3    0.00    0.24     950           ListaHost::insere() [5]
                0.00    0.24     950/950         criaUrl(std::__cxx11::basic_string<char, std::char_traits<ch
                0.00    0.00     950/950         Url::eValida() [890]
                0.00    0.00     950/950         Url::~Url() [891]
```

Deste modo, e preciso reconhecer que o principal ponto de melhoria que o trabalho poderia focar seria uma verificação da validade das *Urls* que evitasse o uso da função `regex_match()`, e em possíveis ajustes para evitar a menor das comparações (uma tarefa desafiadora, pois o autor se esforçou para não inserir muitos aspectos desnecessários).

5.4 Acessos e localidade

Para a análise de acessos e localidade foram retirados os comandos `LISTA_HOSTS`, `ESCALONA_TUDO` e `LIMPA_TUDO` (a estrutura é capaz de limpar tudo no fim do programa), do arquivo gerado `carga1000.txt` (ver seção 5.2) para permitir o acesso e impressão dos elementos.

Infelizmente, o programa `analisamem` utilizado para o processamento dos resultados dos registros de acesso, fornecido pelos professores, não foi capaz de produzir uma saída que o programa `gnuplot` aceitasse, e portanto não foi possível a geração dos gráficos para a análise. O erro encontrado foi o seguinte:

```
$ gnuplot *.gp
Warning: empty y range [0:0], adjusting to [-1:1]
```

```
"escalonador-acesso-1.gp" line 6: warning: Skipping data file with no
valid points
"escalonador-acesso-1.gp" line 6: warning: Skipping data file with no
valid points

plot "escalonador-acesso-0-1.gpdat" u 2:4 w points t "L", "escalonador
-acesso-1-1.gpdat" u 2:4 w points t "E"

"escalonador-acesso-1.gp" line 6: x range is invalid
```

Em termos teóricos, é possível afirmar que como as alocações dos objetos são feitas dinamicamente, a proximidade dos endereços armazenados deve ser consideravelmente diferente de uma alocação estática, e a distância de pilha neste contexto sofrerá um forte impacto. Deste modo, os endereços alocados provavelmente serão encontrados em esparsos locais de memória, sendo o único fator que poderia ajudar nesta situação a "perspicácia" do sistema operacional de fornecer endereços não distantes ao programa quando ele lhe solicitar.

6. Conclusões

Este trabalho contemplou a implementação e documentação de uma estrutura de dado **fila de listas**, ambas encadeadas, para uso em um programa com as funcionalidades de escalonamento de endereços *web*. Percebeu-se que o funcionamento da Fila pouco se distanciava do de uma lista encadeada ordenada, e se verificou que o custo maior no desempenho do programa se caracterizava nem tanto pela estrutura, mas pelas operações que são realizadas sobre as *strings*.

A criação de uma estrutura e programa de porte médio, apesar de trabalhosa e exigir tempo, foi uma útil oportunidade de colocar em prática conceitos vistos na segunda matéria de programação, além de servir como um catalisador para formação de uma base melhor na linguagem C++. Mesmo com planejamento e modelamento feito em antemão, ela ainda apresentou aspectos da implementação que causaram inconveniências e requereram pesquisa e modificação. Este fato do processo de implementação reforçou a percepção da importância do planejamento, e a postura paciente que o programador deve ter ao lidar com os problemas.

A execução do projeto também serviu o propósito de revisar alguns tópicos, como a importância de certos aspectos do programa em projetos maiores; princípios de programação orientada a objetos voltados a classes e herança; o desconfortável uso de variáveis globais em determinadas situações (neste trabalho utilizado para a escrita no arquivo de saída); uso e perigo de variáveis do tipo *unsigned int*; o uso de expressões regulares (*regex*); o uso errôneo de números mágicos e o funcionamento do tipo de dado `string` da biblioteca padrão de C++.

Além disso, também houve o reconhecimento da necessidade de revisão e aprofundamento de algumas práticas, como o manuseio de erros e a possibilidade de se

ter usado o método *try/throw/catch*; a importância da fragmentação da implementação e de se gerar casos de teste (algo que o autor não pode realizar no presente trabalho, mas espera poder fazê-lo no próximo); e uso do termo `const` em métodos e parâmetros.

7. Bibliografia

Bhargava, A. (2016). Hash Tables. In Grokking Algorithms: An illustrated guide for programmers and other curious people. essay, Manning.

Cox, R. (2007, January). Regular Expression Matching Can Be Simple And Fast. Swtch.Com. Retrieved December 19, 2021, from https://swtch.com/%7Ersc/regexp/regexp1.html

Editor, C. S. R. C. C. (n.d.). Host - glossary. CSRC. Retrieved November 26, 2021, from https://csrc.nist.gov/glossary/term/host.

Find out if string ends with another string in C++. (2009, May 17). Stack Overflow. Retrieved December 18, 2021, from https://stackoverflow.com/questions/874134/find-out-if-string-ends-with-another-string-in-c

Guntheroth, K. (2016). Optimized C++. O'Reilly Online Learning. Retrieved December 18, 2021, from https://www.oreilly.com/library/view/optimized-c/9781491922057/ch04.html

Heesch, D. van. (2021, October 21). Special comment blocks. Doxygen Manual: Documenting the code. Retrieved December 15, 2021, from https://www.doxygen.nl/manual/docblocks.html.

How is std::string implemented? (2009, September 23). Stack Overflow. Retrieved December 18, 2021, from https://stackoverflow.com/questions/1466073/how-is-stdstring-implemented

How to remove a particular substring from a string? (2012, May 10). Stack Overflow. Retrieved December 11, 2021, from https://stackoverflow.com/questions/10532384/how-to-remove-a-particular-substring-from-a-string

Kanodia, N., Jacobson, S. (2012). C++ Strings. Stanford CS106B: Programming Abstractions (Handout). Retrieved December 18, 2021, from https://web.stanford.edu/class/archive/cs/cs106b/cs106b.1132/handouts/08-C++-Strings.pdf

Mead, M. (n.d.). Mead's guide to getopt. Mead's Guide To getopt. Retrieved December 1, 2021, from https://azrael.digipen.edu/~mmead/www/Courses/CS180/getopt.html.

Pappa, L. Gisele and Junior, M . Wagner. (2021). Slides virtuais da disciplina de estruturas de dados. Disponibilizado via moodle. Departamento de Ciência da Computação. Universidade Federal de Minas Gerais. Belo Horizonte.

Sanchez, A. (2019). Regex tutorial: Learn with regular expression examples. BreatheCode. Retrieved December 16, 2021, from https://content.breatheco.de/en/lesson/regex-tutorial-regular-expression-examples

Santos, A. S. R. (2013). Learning to Schedule Web Page Updates Using Genetic Programming (dissertation). Belo Horizonte, Minas Gerais. Retrieved December 15, 2021, from https://www.dcc.ufmg.br/pos/cursos/defesas/1612M.PDF.

Verzbickas, A., Mocelin, E. F., Neto, M. B. de S., Siega, R. T. (2013). (report). Relatório Web Crawlers (pp. 3–10). Florianópolis, Santa Catarina. Retrieved December 15, 2021, from https://www.inf.ufsc.br/~luis.alvares/INE5644/G1_WebCrawlers.pdf.

Apêndice: Instruções para compilação e execução

O arquivo `nome_sobrenome_matricula.zip` fornecido em conjunto a este documento deve ser descompactado em um diretório qualquer com uma ferramenta adequada, e deverá conter uma pasta nomeada `TP` e subpastas `src`, `bin`, `obj` e `include`. Caso a estrutura não esteja presente, verifique a procedência do arquivo.

Compilação

Para compilar o programa será necessária a instalação de algumas ferramentas na máquina: o compilador "g++" da GNU Compiler Collection, compatível com versões de C++ a partir de 2017, para a compilação; e do software "make", também da GNU Compiler Collection, para o uso do arquivo "Makefile" (incluso no arquivo do código-fonte).

Em um ambiente Linux (sugere-se o Ubuntu 20.04, da Canonical Ltd.) e a partir do terminal (interface de acesso à linha de comando de sistemas Linux) situado na pasta `TP`, invoque o procedimento de "Makefile" através do comando `make`, que deverá compor uma saída a partir do código fonte e dependências.

O arquivo executável poderá ser encontrado na pasta `bin`, situado dentro da pasta `TP`, com o nome `a.out`.

Execução:

Novamente a partir da linha de comando e de dentro da pasta `TP`, a execução do arquivo deve ser feita através do comando `bin/a.out` (ou de `./a.out` de dentro da pasta `bin`), seguido do nome/endereço do arquivo de entrada (como "entrada.txt", por exemplo, que pode ser colocado na pasta `TP`)e qualquer um dos (ou todos os) parâmetros a seguir:

Parâmetro	Descrição
-m	Ativa o registro de desempenho
-d	Ativa estratégia Breadth-First
-t	Ativa estratégia Best-First
<nome_entrada>	Nome do arquivo a se ler os comandos

Exemplo de comando de execução

bin/a.out entrada.txt -b -m

bin/a.out entrada.txt

bin/a.out entrada.txt -t -m

Formato do arquivo de entrada

O arquivo recebido pelo programa deve possuir uma estrutura bem definida onde cada linha denomina um **comando** ou uma **entrada** (URL). Os comandos disponíveis são denominados a seguir:

1. ADD_URLS `<quantidade>`: adiciona ao escalonador as URLs informadas nas linhas seguintes. O parâmetro `<quantidade>` indica quantas linhas serão lidas antes do próximo comando.
2. ESCALONA_TUDO: escalona todas as URLs seguindo as regras estabelecidas previamente. Quando escalonadas, as URLs são exibidas e removidas da lista.
3. ESCALONA `<quantidade>`: limita a quantidade de URLs escalonadas.
4. ESCALONA_HOST `<host>` `<quantidade>`: são escalonadas apenas URLs deste host.
5. VER_HOST `<host>`: exibe todas as URLs do host, na ordem de prioridade.
6. LISTA_HOSTS: exibe todos os hosts, seguindo a ordem em que foram conhecidos.
7. LIMPA_HOST `<host>`: limpa a lista de URLs do host.
8. LIMPA_TUDO: limpa todas as URLs, inclusive os hosts.

Exemplo de formato de arquivo de entrada

entrada.txt

```
ADD_URLS 3
http://www.youtube.com
http://www.nytimes.com/
http://www.nexojornal.com.br/
ESCALONA 2
LISTA_HOSTS
ADD_URLS 2
http://nytimes.com/world
http://nytimes.com/tech/governance/
ESCALONA_HOST nytimes.com 2
VER_HOST nytimes.com
```