Estudo de caso: Busca em Grafo Aplicada à Gestão de Dependências e Compilação de Projetos em Delphi

Sergio Rocha da Silva
PPGCC
Universidade Federal de São Paulo
São Paulo, Brasil
sergio.rocha.silva@outlook.com

***Abstract*—Projeto de conclusão da disciplina de Análise de Algoritmos e Estruturas de Dados, do programa de pós graduação em Ciência da Computação.**

Esse projeto apresentará um estudo de caso de aplicação das técnicas de busca em grafos (mais especificamente, busca em largura e busca em profundidade), utilizadas no ambiente real de uma instituição particular fornecedora de software ERP, bastante reconhecida no mercado brasileiro, a saber, Nasajon Sistemas.

O caso de uso consiste na gestão de dependências e compilação de projetos Delphi.

***Index Terms*—busca, grafo, largura, profundidade, nasajon, erp, dependências, delphi, compilação**

I. Introdução

No mercado de sistemas de informações, é comum que sistemas complexos sejam desenvolvidos e mantidos durante muitos anos, tendo em vista fatores como estabilidade, durabilidade dos requisitos, disponibilidade financeira das empresas fornecedoras, etc.

Sendo assim, muitos softwares, ainda em uso, foram desenvolvidos com linguagens de programação já consideradas antigas, e que, eventualmente, carecem de recursos comuns para o tempo presente.

E é neste cenário que se concentra o presente estudo de caso, no qual a empresa Nasajon Sistemas, fornecedora de uma solução ERP, passou a enfrentar diversos problemas devido ao constante crescimento de seu código fonte em Delphi (desktop).

Conforme será detalhado no desenvolvimento deste artigo, para contornar as limitações da linguagem de desenvolvimento utilizada (a saber, Delphi XE3), foi necessário lançar mão de diversas refatorações e alterações configuração dos projetos, porém, o problema só foi efetivamente resolvido com a aplicação dos algoritmos de busca em grafo, como núcleo de uma ferramenta de gestão de dependências, desenvolvida sob medida, e cujo projeto foi denominado de nsBuild.

II. Contextualização do Problema

A. Limitações do Delphi XE 3

A versão do produto ERP SQL foi construída, originalmente, com a linguagem de programação Delphi XE3, a qual foi lançada no ano de 2012 pelo fornecedor Embarcadero [1].

Embora contando com diversos recursos avançados de programação (como Generics, Collections, meta programação, suporte nativo a integração com APIs Rest, etc), a linguagem Delphi vem sofrendo um decréscimo contínuo de popularidade (como se pode verificar em fontes como o Tiobe Index [2]), em parte por se tratar de uma linguagem compilada (não gerenciada, com relação ao uso de memória; e compilada diretamente para linguagem de máquina, e não para um formato intermediário, compatível com algum tipo de máquina virtual), e em parte por se tratar de uma linguagem proprietária (de alto custo).

Nesse contexto, o ”ecossistema” da linguagem Delphi, embora não inexpressivo (visto que a mesma ainda figura entre as 20 mais populares do mundo), acabou por se tornar carente de soluções comuns, como gerenciadores de dependências (ferramentas capazes de gerenciar a complexidade gerada pela interdependência entre diferentes projetos, auxiliando etapas como download, construção, e até distribuição de artefatos).

Assim, a arquitetura comum de um sistema Delphi conta, majoritariamente, com os recursos contidos na própria linguagem e, quando necessária a utilização de bibliotecas de terceiros, tais bibliotecas são instaladas ”manualmente” no ambiente de desenvolvimento (uma a uma). Cabendo ao desenvolvedor a garantia de compatibilidade entre as diversas bibliotecas e versões das mesmas.

Além disso, os projetos Delphi, quando se utilizam de módulos externos, ou o fazem por meio de vínculo estático (resolvido em tempo de compilação) ou por meio de vínculo dinâmico, com artefatos de extensão ”.bpl”, que muito se assemelham aos DLLs já comuns no ambiente Windows (sendo que os BPLs se apresentam como um tipo de DLL orientada a objetos).

No entanto, tal tentativa de modularização por meio de BPLs apresenta importantes desafios, se destacando:

- Maior complexidade no uso das dependências (porque é preciso gerenciar, programaticamente, o carregamento das BPLs).
- Perda de performance (pois o vínculo dinâmico das dependências, precisa ocorrer durante a execução dos sistemas).

- Dificuldades com uso de recursos gráficos (notadamente imagens a serem empacotadas num arquivo binário).

Portanto, via de regra, um sistema Delphi não é construído em módulos interdependentes, antes segue contido num único projeto, com vínculo estático para a maioria de suas dependências.

B. Crescimento Acelerado do Código Fonte

Pelas razões já apresentadas, sistemas Delphi tendem a crescer continuamente em quantidade de linhas de código.

No caso do ERP Nasajon, podemos facilmente evidenciar tal crescimento no gráfico a serguir, o qual representa o crescimento acumulado de linhas de código num período de um ano e meio (entre maio de 2017 e novembro de 2018):

Fig. 1. Crescimento acumulado de linhas de código

C. Problemas do Compilador Delphi

Tendo em vista esse crescimento acelerado, chegando à marca do acréscimo de quase 2.5 milhões de linhas em um ano e meio, é fácil ver que o ERP Nasajon rapidamente forçou os limites do compilador Delphi.

E, de fato, o compilador passou a apresentar recorrentemente dois tipos de problema:

- Estouro de memória (F2046 Out of memory).
  - Causa do problema, de acordo com o fornecedor da linguagem (conforme encontrado no site do mesmo): ”You get this error when the RAD Studio built-in compiler runs out of memory. This is a rare error that might occur when you build an extremely large project group of applications and libraries.” [3]
- Erros internos do compilador, de natureza desconhecida (F2084: Internal Error).
  - Causa do problema, de acordo com o fornecedor da linguagem (conforme encontrado no site do mesmo): ”This error message indicates that the compiler has encountered a condition, other than a syntax error, that it cannot successfully process. Sometimes, this error is because the memory consumption at compile time is approaching the limit of the IDE during the build.” [4]

III. Tentativas de Solução

A. Compilação Externa

De acordo com a documentação do fornecedor da linguagem, a única sugestão dada para contornar os problemas apresentados na compilação do sistema, seria a compilação externa dos projetos Delphi, isto é, a compilação direta por linha de comando, e não por meio da IDE.

Esta solução foi adotada por um largo período, no entanto, além de apresentar alguns reveses (como perda de performance, devido à abertura de processos externos de compilação, a cada simples teste por parte dos programadores), acabou por ser também uma solução de caráter temporário, visto que os mesmos problemas de compilação são apresentados (ainda que mais tardiamente) na compilação externa.

B. Divisão em Projetos Menores

Após consumidos os poucos recursos liberados pela compilação externa, a segunda tentativa de solução consistiu na quebra dos projetos Delphi em projetos menores, interdependentes.

Tais projetos são compilados em sequência, gerando artefatos binários que, após vínculo estático (link), se tornarão em executáveis (isto é, são gerados arquivos com extensão ”.dcu”, que é equivalente à extensão ”obj”, comum no contexto da linguagem C).

No entanto, para manter a coesão entre tais pacotes, foi preciso definir escopos coerentes, uma vez que a compilação com referência circular não é possível em Delphi.

Assim, o número de subprojetos passou a crescer rapidamente, levando ao novo desafio da gestão de dependências, uma vez que, como já dito nas sessões inicias, o ecossistema Delphi não apresenta boas ferramentas destinadas a esta finalidade.

O resultado desta tentativa foi uma longa fila de projetos a serem compilados numa ordem fixa, dificultando muito qualquer tipo de implementação que exigisse adição de artefatos intermediários na fila (pois é preciso garantir que cada projeto seja compilado após todas as suas dependências, e antes daqueles que o utilizem).

Obs.: Para o leitor familiarizado com o Delphi, o recurso dos grupos de projetos pode ser lembrado a esta altura. A saber, tal recurso foi também utilizado, chegando a haver cerca de 20 grupos de projeto, cada um com dezenas projetos, o que tornou a gestão de dependências muito manual e complexa).

IV. Solução do Problema

A. Modelagem em Grafo

Conforme apresentado, o problema de construção dos projetos Delphi, pode ser facilmente modelado por meio de um ”Grafo Acíciclo Dirigido”, uma vez que cada projeto pode ser visto como um vértice, e cada dependência como uma aresta.

Como se pode ver no pequeno recorte da figura 2, o grafo de dependência entre os projetos representa com precisão os projetos Delphi, e suas dependências.

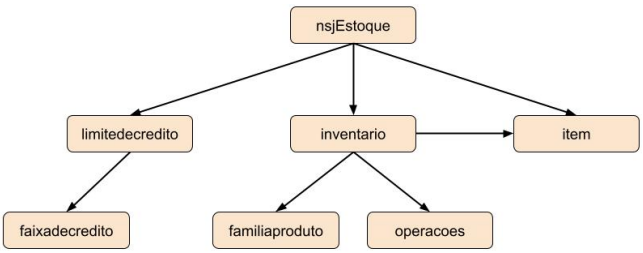

Fig. 2. Recorte da modelagem em grafo

A título de exemplo, segue breve descritivo dos projetos acima destacados:

- nsjEstoque: Projeto que representa o módulo reponsável pela gestão de Estoque (no contexto do ERP como um todo).
- limitedecredito: Pacote que contém as classes responsáveis pela gestão dos limites de crédito (fornecidos aos clientes).
- inventario: Pacote que fornece soluções de inventário do estoque (apuração de saldo real dos itens).
- item: Pacote responsável pelo CRUD de itens de estoque (produtos e materiais).
- faixadecredito: CRUD das faixas de crédito passíveis de atribuição aos clientes.
- familiaproduto: CRUD das famílias de produto (tipo de agrupamento).
- operacoes: CRUD dos grandes grupos de configuração das movimentações de estoque possíveis.

É importante notar que tal grafo só pode ser interpretado como uma árvore devido sua característica de ser dirigido, uma vez que a definição mais simples de árvores, implica em grafos conexos e acíclicos.

Tais grafos de dependências são conexos, e, embora aparentem ciclos, não podem apresentar dependência circular (por conta das características da linguagem Delphi).

Sendo assim, a modelagem do problema consiste em árvores de dependências (se considerados todos módulos do ERP ao mesmo tempo - por exemplo, Estoque, Finanças, etc -, tem-se uma modelagem em forma de uma floresta de dependências).

B. Busca em Profundidade

Resumidamente, o resultado desejado é a compilação do ERP como um todo (isto é, de todos os projetos contidos no grafo).

Para se alcançar esta compilação, é necessário que os projetos sejam construídos em uma ordem específica, que caminhe desde às folhas, até a raiz das árvores, garantindo que um vértice só entre na lista após suas próprias dependências.

O nome deste tipo de ordenação é ”Ordenação Topológica”, que, conforme definição:

> DEFINIÇÃO: A ordenação topológica de um dag (directed acyclic graph) G= (V, E) é uma ordem linear (sequência, lista) de vértices tal que se G contém um arco (u,v) então u aparece antes de v na ordem linear. Claramente, se o grafo for cíclico, tal ordem não existe. [5]

E, tal tipo de ordenação pode ser alcançado por meio da Busca em Profundidade, bastando que o algoritmo seja adaptado para retornar uma lista, onde os vértices são adicionados a cada momento em que um destes é terminado (ou pintado de preto, dependendo da implementação escolhida).

Além disso, também pode-se adaptar a busca em profundidade, para lançar uma exceção no caso de se encontrar um vértice já visitado durante o apronfundamento no grafo. Isto porque, se um vértice já começou a ser explorado, e é visitado novamente, logo se alcançou uma condição de dependência circular, o que não pode ser resolvido no problema de compilação para a linguagem Delphi.

Portanto, o núcleo da solução desenvolvida consiste justamente na utilização da busca em profundidade.

C. Paralelismo

Outra característica importante do problema, é que, embora haja uma ordem topológica para a solução, tal ordem não é única.

Resumidamente, a escolha da aresta a ser utilizada no aprofundamento da busca não prevê nenhum tipo de ordem, e portanto tem caráter aleatório.

Assim, facilmente, pode-se incluir projetos independentes, porém sequenciais na fila de ordenação. Caso em que pode-se disparar a compilação paralela entre tais projetos.

Sendo assim, para bom aproveitamento dos recursos de hardware da máquina responsável pela compilação, a solução técnica deve prever o disparo de compilações paralelas (para esses casos de enfileiramento de vértices independentes).

D. Construção Parcial e Busca em Largura

Considerando o cenário em que todos os projetos já tenham sido compilados, é fácil ver que, uma alteração num deles, invalidaria seu próprio status de compilação, e, em sequência, o status de todos os projetos que dependam do mesmo.

Considere o seguinte grafo de exemplo (onde os vértices em azul estão compilados, e os em branco foram recém alterados):

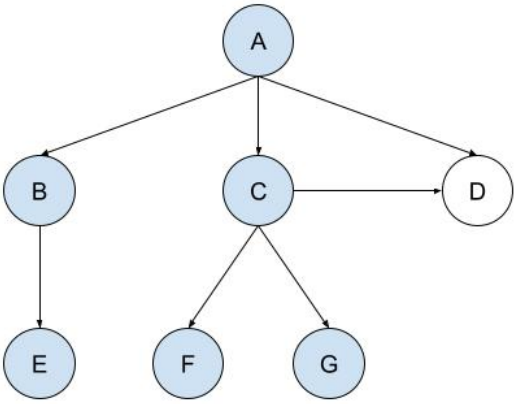

Fig. 3. Exemplo de status de compilação recém alterado

Se for invertido o direcionamento das arestas, e executada uma busca em largura a partir do vértice ”D”, pode-se marcar os vértices alcançados pela busca com o status de ”não compilados”.

O que levaria ao seguinte novo grafo:

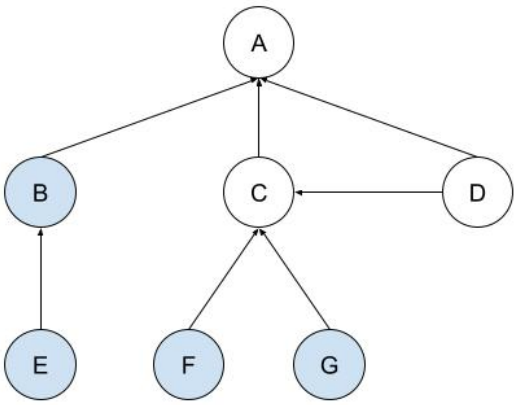

Fig. 4. Mudando status de compilação pela busca em largura

Portanto, com uma pequena adaptação na interpretação do grafo, é possível utilizar o algoritmo da busca em largura para marcar os vértice pendentes de compilação, e em seguida pode-se rodar uma nova busca em profundidade, a partir da raiz, compilando novamente apenas os vértices necessários (o que representa uma boa otimização de tempo no uso prático da ferramenta, uma vez que pode-se compilar apenas o que for necessário, a partir de uma alteração num ponto qualquer do grafo de projetos).

V. Detalhes de Implementação

A. Organização do Projeto

O projeto resultante deste esforço está disponível como OpenSource no GitHub da Nasajon Sistemas. [6]

Tal projeto foi desenvolvido em linguagem Java, e está estruturado como um utilitário de linha de comando, contando com diversas funcionalidades, das quais se destacam:

- update (default): Compilação de um determinado projeto, considerando apenas os projetos não compilados na árvore (foi desenvolvido um mecanismo de detecção dos projetos alterados, que não carece de ser aqui detalhado, mas que se baseia na data de alteração dos arquivos que compõe os projetos).
- validate: Ferramenta capaz de verificar a consistência do grafo de dependências, baseado na interpretação dos arquivos de declaração dos projetos Delphi (isto é, verifica se as dependências declaradas estão de acordo com o código fonte dos projetos, indicando ajustes).
- clean: Limpa a cache de compilação, forçando uma compilação completa dos projetos.

Sendo também útil explicar as principais partes da árvore de diretórios do projeto:

- nsjBuild/src/br/com/nasajon/nsjbuild (Raiz do fonte do projeto)
  - controller (Diretório de classes de controle das buscas)
    - BuscaLargura.java (Implementação da busca em largura)
    - BuscaProfundidade.java (Implementação da busca em profundidade)
    - ControleCompilacao.java (Interface que abstrai o controle de chamadas ao compilador Delphi)
    - Compilador.java (Classe que implementa a interface de compilação)
    - ThreadCompilacao.java (Thread para suportar o paralelismo de compilação)
  - delphi (Classes para integração com o ambiente Delphi)
  - exception (Classes para controle de exceções)
  - model (Classes para abstração da representação em grafo)
  - modelXML (Classes para abstração da representação dos grafos, em arquivos XML, comentado mais à frente)
  - util (Utilitários diversos)

B. Cenário Prático

Na prática, a modelagem do problema em grafos, se deu por meio da representação dos projetos em formato XML.

A título de exemplo, considere o XML do projeto ”limitedecredito” já citado nos exemplos anteriores:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<projeto xmlns="http://www.nasajon.com.br/nsjbuild"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.nasajon.com.br/nsjbuild␣
      nsjBuildProject.xsd␣">
    <nome>limitedecredito</nome>
    <autor>marcelocouto</autor>
    <dataCriacao>21/06/2021</dataCriacao>
    <resumo>Pacote que contem as classes responsaveis pela
         gestao dos limites de credito (fornecidos aos
         clientes)</resumo>
    <path>commonfeature\limitedecredito\package\
         limitedecredito.dproj</path>
    <dependencias>
        <dependencia>limitedecreditohistorico</dependencia>
        <dependencia>faixadecredito</dependencia>
        <dependencia>nsjusuarios</dependencia>
        <dependencia>entidadesempresariais</dependencia>
    </dependencias>
</projeto>
```

Note que:

- O projeto é descrito por meio das tags: nome, autor, dataCriacao e resumo.
- Logo a seguir, tem-se uma listagem de dependências, indicando os projetos do qual o atual depende, o que permite a construção das arestas do grafo.

A título de exemplo, para o módulo ”nsjEstoque” do ERP, foi necessária de definição de 249 XMLs, conforme figura 5.

Portanto, por meio da simples linha de comando a seguir, o nsBuild irá se responsabilizar pela construção do grafo de dependências (modelagem em grafo), verificação dos projetos a serem realmente compilados (busca em largura), ordenação topológica (busca em profundidade) e compilação, inclusive com paralelismo (integração com o ambiente Delphi):

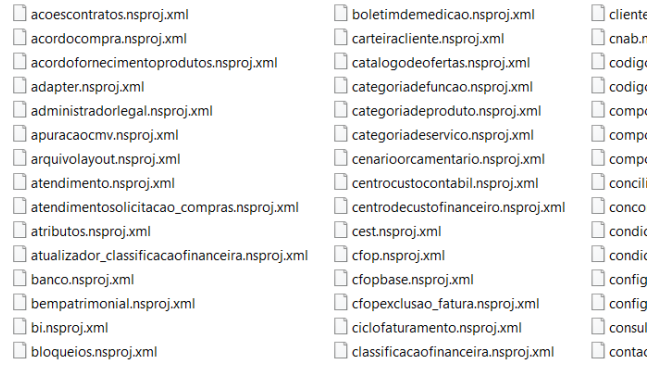

Fig. 5. Recorte da listagem de xmls do projeto nsjEstoque

```
> nsbuild nsjEstoque
```

VI. Resultados

A. Análise Teórica (complexidade)

O custo de execução da ferramenta consiste, de três etapas:

1) Montagem do grafo.
2) Busca em largura.
3) Busca em profundidade.

Em se tratando de etapas sequenciais, o custo total é apenas o somatório dos custos individuais.

Para o cálculo da complexidade, considere ”n” o número de vértices (projetos), e ”m” o número de arestas (dependências).

A primeira etapa, de criação do grafo, consiste na leitura dos XMLs dos projetos, inserindo-os no grafo, bem como suas respectivas dependências, o que resulta numa complexidade: O(n+m).

Já as duas etapas seguintes, conforme literatura, também apresentarão complexidade O(n+m) [7] e [8].

Logo, resumindo a complexidade das etapas:

Complexidade das Etapas	
Montagem do grafo	O(n)
Busca em largura	O(n+m)
Busca em profundidade	O(n+m)

E, portanto, a complexidade total será aproximadamente O(3n+3m), ou O(n+m).

B. Análise de Performance

A performance real da compilação total não é de fato melhorada pelo uso das técnicas de busca em grafo, porque o número total de projetos a compilar é sempre o mesmo.

No entanto, a performance é positivamente impactada pelo uso do paralelismo de compilação, para o que, segue breve comparativo:

Comparação de performance da compilação total	
Sem paralelismo	51 minutos
Máximo de 2 projetos simultâneos	32 minutos
Máximo de 3 projetos simultâneos	25 minutos

Além disso, outro grande ganho no uso da ferramenta está no fato de que não é preciso compilar sempre todos os projetos, a cada alteração de código. No entanto, tal melhoria de performance é mais difícil de demonstrar, pois, no pior caso, tem-se a compilação total, e, na prática, dependerá do projeto alterado pelo usuário.

VII. Conclusão

Este estudo de caso demonstrou que a modelagem de um problema real em grafos, com a devida aplicação das técnicas de busca (em largura, e em profundidade), pôde impactar positivamente a produtividade de uma companhia fornecedora de sistemas de informação, mesmo em cenários de utilização de tecnologia legada. Servindo de incentivo para melhor aplicação de algoritmos consagrados na teoria da computação, mesmo quando tais recursos não se encontram já disponíveis no ecossistema de uma linguagem qualquer.

References

[1] Delphi (software). Wikipedia, 24, abril de 2023. Desenvolvimento. Disponível em: https://pt.wikipedia.org/wiki/Delphi_(software). Acesso em: 23, 06 de 2023.

[2] TIOBE Index for June 2023. Tiobe, junho de 2023. Disponível em: https://www.tiobe.com/tiobe-index/. Acesso em: 23, 06 de 2023.

[3] F2046 Out of memory (Delphi). Embarcadero, 25, Maio de 2016. Disponível em: https://docwiki.embarcadero.com/RADStudio/Alexandria/en/F2046_Out_of_memory_(Delphi). Acesso em: 28, 06 de 2023.

[4] F2084 Internal Error. Embarcadero, 25, Maio de 2016. Disponível em: https://docwiki.embarcadero.com/RADStudio/Alexandria/en/F2084_Internal_Error_-_%25s%25d_(Delphi)). Acesso em: 28, 06 de 2023.

[5] PROLO, Carlos Augusto. Ordenação Topológica de Grafos. UFRN. Definição. Disponível em: https://www.dimap.ufrn.br/~prolo/Disciplinas/13I/DIM0111.0-AEDII/materiais/grafos/07%20Grafos%20Dirigidos%20--%20Ordenacao%20Topologica.pdf. Acesso em: 23, 06 de 2023.

[6] nsBuild. Nasajon. Disponível em: https://github.com/Nasajon/nsbuild. Acesso em: 23, 06 de 2023.

[7] MANDEL, Arnaldo. DFS e ordenação topológica. IME-USP. Consumo de tempo. Disponível em: https://www.ime.usp.br/~am/5711/aulas/aula15h.pdf. Acesso em: 23, 06 de 2023.

[8] Busca em largura. Wikipedia. Complexidade de Tempo. Disponível em: https://pt.wikipedia.org/wiki/Busca_em_largura. Acesso em: 23, 06 de 2023.