Um Método *Branch & Bound* para Maximizar o Valor de Projetos de *Software* para o Negócio

Antonio Juarez Alencar, Eber Assis Schmitz, Ênio Pires de Abreu

[1]Departamento de Ciência da Computação e
Núcleo de Computação Eletrônica
Universidade Federal do Rio de Janeiro
Caixa Postal 68.530, 21941-590 – Rio de Janeiro – RJ, Brasil

juarezalencar@br.inter.net, eber@nce.ufrj.br, enioabreu@gmail.com

Abstract. *This article presents a branch & bound approach that allows software managers to determine the optimum order for the development of a network of dependent software parts that have value to customers. In many different circumstances the method allows for the development of complex and otherwise expensive software from a relatively small investment, favoring the use of software development as a means of obtaining competitive advantage.*

Resumo. *Este artigo apresenta uma abordagem branch & bound que permite aos gerentes de projeto determinar a melhor ordem de desenvolvimento de uma rede de módulos interdependentes de* software *que possui valor para os clientes. Em diversos casos, o método permite o desenvolvimento de software complexos e caros a partir de investimentos relativamente pequenos, favorecendo o desenvolvimento de software como meio para a obtenção de vantagem competitiva.*

1. Introdução

Nos mercados altamente competitivos e globalizados que marcam este início de século, é improvável que projetos de *software* que não apresentem riscos conhecidos e aceitáveis para o negócio sejam sequer considerados para desenvolvimento [McManus 2003]. Nestes mercados, os investidores clamam, cada vez mais e mais alto, por um rápido retorno de seus investimentos, períodos mais curtos para o desenvolvimento e comercialização de produtos, e uma arquitetura organizational ágil e flexível [Lam 2004, Helo et al. 2004, Whittle and Myrick 2005]. Tudo isso requer o uso de novas abordagens nos projetos de desenvolvimento de *software* e ferramentas capazes de reduzir custos, agilizar processos e melhorar a performance de produtos e serviços [Jorgenson et al. 2003, Highsmith 2002].

Para lidar adequadamente com esta situação, tanto acadêmicos quanto desenvolvedores de *software* têm enfatizado a necessidade de métodos, conceitos e ferramentas que favoreçam a entrega rápida de funcionalidades que tenham valor para os clientes [Abacus et al. 2005, Nord and Tomayko 2006]. Neste contexto, uma abordagem de enfoque financeiro para a priorização de funcionalidades denominada *Incremental Founding Method*, ou IFM, surgiu como uma forma de aumentar o valor potencial dos projetos de *software* [Denne and Cleland-Huang 2004a, Denne and Cleland-Huang 2004b].

No IFM, as funcionalidades do *software* são agrupadas em unidades auto-contidas que criam valor para os negócios em uma ou mais das seguintes áreas:

- *Criação de diferencial competitivo* - a unidade de *software* permite a criação de serviços e produtos que tem valor para os clientes e que são diferentes de tudo que é oferecido no mercado;
- *Aumento do faturamento* - embora a unidade de *software* não ofereça nenhuma nova funcionalidade para os clientes, ela ajuda a aumentar o faturamento ao permitir que produtos com qualidade equivalente a dos concorrentes sejam oferecidos por um preço menor;
- *Redução de custos* - a unidade de *software* faz com que um ou mais processos de negócio tenham seu custo de execução reduzidos;
- *Projeção da marca* - ao construir a unidade de *software* a empresa se projeta como sendo tecnologicamente avançada e
- *Aumento da fidelidade dos clientes* - a unidade de *software* faz com que os clientes comprem mais, mais freqüentemente ou ambos.

Obviamente, o valor total trazido para uma organização por um *software* constituído de várias unidades interdependentes é fortemente influenciado pela ordem de implementação destas unidades, dado que cada uma possui seu próprio fluxo de caixa e restrições de precedência. Por este motivo, o IFM inclui um conjunto de estratégias de tempo polinomial que identificam um plano de desenvolvimento que aumenta o valor dos projetos, reduzindo os investimentos iniciais ou melhorando outras métricas de projeto tais como: o tempo necessário para se atingir o ponto de equilíbrio e o tempo de retorno do investimento [Denne and Cleland-Huang 2005].

No entanto, o IFM é um método aproximativo, portanto nem sempre as estratégias identificadas pelo método levam ao melhor plano possível que, no caso geral, só pode ser descoberto em tempo exponencial. Além disso, o IFM requer que cada unidade de *software* dependa de no máximo uma outra, para que o plano seja encontrado em tempo polinomial.

Este artigo apresenta um método *branch & bound* para encontrar um plano de desenvolvimento que maximize o valor de um projeto de *software* para o negócio. Este método sempre encontra a solução ótima e não impõe restrições às relações de dependência que possam existir entre as unidades do *software*, o que o torna mais adequado às diversas estruturas de interdependência que ocorrem no mundo real. Embora o método seja exponencial, em muitas circunstâncias ele pode encontrar a melhor solução para o problema em tempo polinomial. Veja [Liberti 2003, Hillier and Lieberman 2001] para uma introdução aos métodos *branch & bound*.

O restante deste artigo está organizado da seguinte forma. Os fundamentos conceituais necessário a compreensão do método são introduzidos na Seção 2. Na Seção 3 o método é apresentado com o auxílio de um exemplo inspirado no mundo real. A formalização do método é feita na Seção 4. A Seção 5 discute diferentes aspectos das implicações do desenvolvimento do método. As conclusões são apresentadas na Seção 6.

2. Fundamentos Conceituais

O IFM preconiza o particionamento dos projetos de *software* em unidades auto-contidas denominadas *minimum marketable features*, ou MMFs. Esses pacotes agrupam funcionalidades que possam ser entregues em conjunto e que tenham valor para os clientes [Steindl 2005, Denne and Cleland-Huang 2004b]

Apesar de todo MMF ser auto-contido, são comuns os casos em que um MMF só pode ser desenvolvido depois que outras partes do projeto tenham sido concluídas. Estas outras partes podem ser outros MMFs ou a infra-estrutura arquitetural, isto é, o conjunto dos ítens básicos do projeto que não oferecem nenhum valor direto aos clientes, mas são requeridos pelos MMFs. Por exemplo, a biblioteca de interfaces gráficas que permite que os diversos módulos que compõem um *software* tenham a mesma identidade gráfica não têm nenhum valor direto para os clientes, entretanto o bom senso nos indica que nenhuma unidade de *software* deveria ser desenvolvida até que ela tenha sido criada.

A própria infra-estrutura arquitetural pode ser dividida em unidades auto-contidas passíveis de serem entregues separadamente. Estes elementos chamados de *architectural elements*, ou AEs, permitem que a infra-estrutura seja entregue conforme a necessidade, reduzindo o investimento inicial necessário para o desenvolvimento do projeto.

2.1. Fluxo de Caixa

Depois que os MMFs e os AEs são identificados, desenvolvedores e especialistas do negócio trabalham em conjunto para analisar cada MMF e AE, estimando os custos e os ganhos esperados para cada unidade no decorrer de uma janela de oportunidade. Estes custos e ganhos formam um fluxo de caixa que pode ser utilizado para estimar o valor total do *software*. Por exemplo, a Figura 1 apresenta um conjunto de MMFs e AEs de um projeto de *software* que dá suporte a atividades de marketing direto. Os nós *Start* e *End* servem respectivamente para marcar o começo e o término do projeto, possuindo duração, custo e valor zero. A Tabela 1 identifica cada uma destas unidade de *software* e mostra os seus respectivos fluxos de caixa esperados.

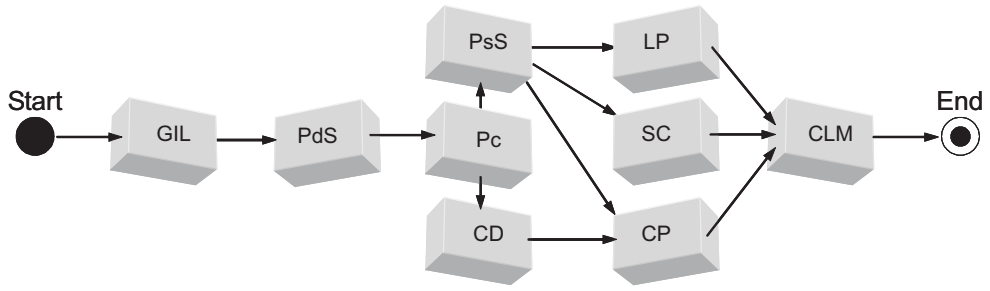

Figure 1. A rede de atividades de um projeto de *software*

Observe que, na Tabela 1, o tempo decorrido entre o primeiro e o último período é a janela de oportunidade do projeto, isto é, o intervalo de tempo no qual o projeto tem valor para o negócio. Em termos formais, uma janela de oportunidade P é um conjunto $\{p_1, p_2, ..., p_k\}$ de períodos de mesma duração. Neste caso, $P = \{1, 2, ..., 12\}$. O fluxo de caixa de uma unidade de projeto v é dado por $fc(v)$ e o elemento do fluxo de v no período $p \in P$ é dado por $fc(v, p)$. Na Tabela 1, $fc(CD)$ no período 1, ou $fc(CD, 1)$, é -70. Por sua vez, a rede de atividades de um projeto de *software* é um grafo direcionado acíclico $G(V_G, E_G)$, no qual:

- $V_G = \{v_1, v_2, \cdots, v_n\}$ é um conjunto de MMFs e AEs, onde cada uma das unidades do *software* possui uma duração prevista $d(v_i)$ e um fluxo de caixa associado $fc(v_i)$ avaliado para uma janela de oportunidade P, e
- E_G é um conjunto de pares ordenados, tal que se $(v_a, v_b) \in E_G$, então v_b depende de v_a.

Unidades do Projeto	Tipo da Unidade	Atividade	Períodos				
			1	2	3	···	12
GIL	AE	Graphical Interface Library	-50	0	0	···	0
PdS	MMF	Product Selection	-40	20	20	···	20
PsS	MMF	Prospect Selection	-50	30	30	···	30
Pc	MMF	Pricing	-30	15	15	···	15
CD	MMF	Catalog Design	-70	20	20	···	20
LP	MMF	Label Printing	-20	5	5	···	5
SC	MMF	Stock Control	-200	40	40	···	40
CP	MMF	Catalog Printing	-50	15	15	···	15
CLM	MMF	Catalog Labeling & Mailing	-50	200	200	···	200

Table 1. Os Fluxos de Caixa das unidades do projeto em milhares de dólares americanos.

No grafo apresentado na Figura 1, temos $V_G = \{Start, GIL, PdS, \cdots, CLM, End\}$ e $E_G = \{(Start, GIL), (GIL, PdS), (PdS, Pc), \cdots, (CP, CLM), (CLM, End)\}$. Uma boa introdução à teoria de grafos pode ser encontrada em [Gross and Yellen 2005].

2.2. Fluxo de Caixa Descontado

Para que se possa comparar os valores de diferentes MMFs no decorrer do tempo é preciso encontrar seus fluxos de caixa descontado (FCDs), já que não é apropriado executar operações matemáticas em valores monetários em diferentes instantes de tempo sem considerar uma taxa de juros [Fabozzi et al. 2006]. A soma de um FCD é o seu valor presente líquido, ou VPL. Em termos formais, o VPL de uma unidade de *software* v, cujo desenvolvimento inicia no período $t \in P$ é dado por

$$vpl(v,t) = \sum_{j=t}^{n} \frac{fc(v, j-t+1)}{(1+\frac{r}{100})^j},$$

onde r é a taxa de desconto e n é o último período da janela de oportunidade P. Por exemplo, se o desenvolvimento do MMF *CD* começar no período 1, então seu valor presente líquido é dado por

$$vpl(CD, 1) = \frac{-70}{(1+\frac{2}{100})^1} + \frac{20}{(1+\frac{2}{100})^2} + \ldots + \frac{20}{(1+\frac{2}{100})^{12}} = \$123$$

A Tabela 2 mostra o VPL de cada MMF da Figura 1, considerando o início do desenvolvimento em diferentes pontos da janela de oportunidade P, a uma taxa de desconto de 2%. Note que a tabela só considera os nove primeiros períodos da janela de oportunidade, já que são nestes períodos que as unidades de software serão construídas.

2.3. Valor Presente Líquido de um Projeto

Naturalmente, o valor de um projeto depende da ordem em que as unidades do *software* são produzidas. Por exemplo, se as unidades de *software* da Figura 1 forem desenvolvidas na ordem *GIL* $\rightarrow$ *PdS* $\rightarrow$ *Pc* $\rightarrow$ *CD* $\rightarrow$ *PsS* $\rightarrow$ *SC* $\rightarrow$ *CP* $\rightarrow$ *LP* $\rightarrow$ *CLM*, então o retorno obtido será de

Unidades do Projeto	Períodos								
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
GIL	-49	-48	-47	-46	-45	-44	-44	-43	-42
PdS	153	134	116	98	81	64	47	31	15
PsS	239	211	184	157	131	105	80	55	31
Pc	115	101	87	74	61	48	35	23	11
CD	123	105	88	71	54	37	21	5	-10
LP	28	24	20	15	11	7	3	-1	-5
SC	188	153	119	86	53	21	-10	-41	-71
CP	95	81	68	55	43	30	18	6	-6
CLM	1870	1679	1491	1307	1127	950	777	607	441

Table 2. VPLs das unidades considerando uma taxa de 2% por período.

$$vpl(\textit{GIL}, 1) + vpl(\textit{PdS}, 2) + \cdots + vpl(\textit{CLM}, 9) = -\$49 - \$134 + \cdots + \$441 = \$853$$

No entanto, se a ordem de desenvolvimento for *GIL* $\rightarrow$ *PdS* $\rightarrow$ *Pc* $\rightarrow$ *CD* $\rightarrow$ *PsS* $\rightarrow$ *LP* $\rightarrow$ *SC* $\rightarrow$ *CP* $\rightarrow$ *CLM*, então o retorno será de \$818.

Em termos financeiros, a soma dos valores presente líquidos de uma seqüência S de MMFs e AEs correspondente a um dado projeto G é o "valor presente líquido de S", ou *vpl(S)*. Em linguagem formal

$$\textit{vpl}(S) = \textit{vpl}(s_1, 1) + \sum_{i=2}^{|S|} \textit{vpl}(s_i, 1 + \sum_{j=1}^{i-1} d(s_j))$$

onde $S = s_1, s_2, \cdots, s_m$ é uma seqüência de unidades de *software* pertencentes a V, $|S|$ é o número de unidades de *software* na seqüência, i é o período no qual s_i é desenvolvido e $d(s_j)$ é a duração do desenvolvimento da unidade de software s_j.

2.4. Métodos Branch & Bound

De acordo com [Liberti 2003] os métodos *branch & bound* encontram-se entre os mais bem sucedidos e os mais utilizados na solução de problemas de otimização não linear. Se o tamanho e a complexidade do problema que se deseja resolver dificultam a solução direta, divide-se o problema sucessivamente em subproblemas cada vez menores até que apresentem tamanho e complexidade passíveis de solução direta. Portanto, dividir para conquistar é o conceito básico por trás desta família de métodos

A divisão (*branch*) é feita particionando-se o conjunto de soluções válidas em subconjuntos cada vez menores. A conquista é feita calculando-se um limite (*bound*) de quão boa a melhor solução do subconjunto pode vir a ser. Subconjuntos são descartados quando seus limites indicam que não existe a possibilidade deles conterem uma solução ótima para o problema original. Esta estratégia nos leva a um método composto por dois passos que geram uma árvore de busca para encontrar a solução ótima. Enquanto o passo do *branch* é responsável por fazer com que a árvore cresça, o passo do *bound* é responsável por limitar esse crescimento [Hillier and Lieberman 2001].

3. Um Estudo de Caso

Considere a existência de uma cadeia de lojas de móveis que tem como prática o envio de catálogos para aumentar suas vendas. Em uma operação padrão, a empresa monta o catálogo com uma variedade de produtos pré-selecionados e o envia para um grupo de clientes potenciais, que são obtidos nos bancos de dados da empresa. Para obter um resultado satisfatório, esta tarefa requer a execução de oito atividades em um curto intervalo de tempo, a saber

1. *Product Selection* – seleciona quais produtos serão anunciados no catálogo;
2. *Prospect Selection* – identifica os clientes potenciais para os quais os catálogos devem ser enviados;
3. *Pricing* – estabelece o preço promocional de todos os produtos que serão anunciados no catálogo;
4. *Catalog Design* – o aspecto gráfico e textual do catálogo, e a presença de outros materiais promocionais, são concebidos e reunidos;
5. *Label Printing* – as etiquetas contendo os nomes e endereços das pessoas que fazem parte do público alvo são impressas e organizadas;
6. *Stock Control* – a disponibilidade dos produtos anunciados no catálogo, para que sejam entregues quando solicitados;
7. *Catalog Printing* – o catálogo é realmente impresso;
8. *Catalog Labeling & Mailing* – os catálogos são etiquetados com os nomes e endereços das pessoas que fazem parte do público alvo, e depois enviados para seus destinos pelo correio.

Com o objetivo de aumentar a eficiência das campanhas de marketing por catálogo, a companhia decidiu desenvolver um sistema baseado em ferramentas de *software* que, atuando em conjunto, fornecem o suporte adequado às atividades executadas durante as campanhas. Como cada atividade é apoiada por uma ferramenta diferente, um total de oito ferramentas deverão ser construídas de forma que as informações disponibilizadas por cada uma possam ser utilizadas pelas outras.

A Figura 1 apresenta a rede de atividades relativa ao desenvolvimento das ferramentas de software. A biblioteca da interface gráfica (*GIL*) é um elemento arquitetural que disponibiliza os componentes utilizados para criar uma identidade visual única para todas as interfaces do *software*. A Tabela 1 mostra o investimento necessário e os ganhos esperados de cada ferramenta, enquanto que a Tabela 2 apresenta o valor que o desenvolvimento de cada ferramenta trará para a empresa.

A competição acirrada no negócio de móveis fez com que as margens de lucro da empresa caíssem nos últimos anos. Assim sendo, devido à falta de recursos apropriados, apenas uma ferramenta poderá estar em desenvolvimento a cada instante de tempo. Diante deste cenário, a empresa decidiu que todas as propostas de desenvolvimento de novos *softwares* devem estar acompanhadas de um relatório discriminando a necessidade de cada módulo, juntamente com as estimativas de custo e retorno esperados.

Ciente destes critérios e objetivando aumentar as chances de aprovação, o gerente responsável pelo projeto em questão decidiu que as ferramentas que compõem o sistema devem ser desenvolvidas de forma que o retorno proporcionado pelo projeto seja máximo. Como o retorno financeiro do projeto é altamente dependente da ordem em que

as unidades de *software* são desenvolvidas, é preciso encontrar a ordem de desenvolvimento que maximiza o VPL do projeto.

3.1. Geração da Árvore de Busca

Para encontrar essa ordem, optou-se por criar uma árvore de busca utilizando o método *branch & bound*. A seqüência escolhida será aquela que apresentar um VPL maior que todas as outras seqüências representadas na árvore.

Passo 1: *Inicialização*

A construção da árvore é iniciada com a inserção do nó *Start* na sua raiz. A esse nó atribui-se o identificador zero. A Figura 2 mostra a composições do nó zero.

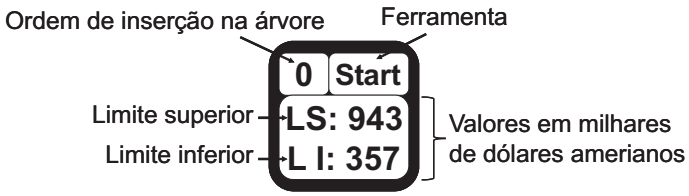

Figure 2. Estado da árvore após a inicialização da busca.

Observe que o limite superior do nó é a soma dos VPLs de cada unidade de *software* (MMF ou AE) da parte conhecida da seqüência com os VPLs máximos de cada unidade da parte desconhecida. Neste caso, como o nó zero é o único nó da árvore, o limite superior do nó é o somatório dos maiores VPLs de cada unidade de *software* ao longo da janela de oportunidade, respeitada as precedência apresentadas na Figura 1. Portanto, de acordo com as informações apresentadas na Tabela 2,

$$ls(0) = vpl(GIL, 9) + vpl(PdS, 2) + vpl(PsS, 4) + vpl(Pc, 3) + vpl(CD, 4) +$$

$$vpl(LP, 5) + vpl(SC, 5) + vpl(CP, 6) + vpl(CLM, 9)$$

$$= -42 + 134 + 157 + 87 + 71 + 11 + 53 + 30 + 441 = 943$$

Note que, para efeito do cálculo de $ls(0)$, o maior elemento do fluxo de caixa descontado de *Pc* é \$87, e não \$115 como poderia se supor. Isso ocorre porque as restrições de precedência só permitem que *Pc* seja desenvolvido a partir do terceiro período. Ver Figura 1.

O limite inferior do nó zero é o somatório dos menores VPLs de cada unidade de *software* ao longo da janela de oportunidade, a partir do período mais cedo em que cada uma possa ser desenvolvida, considerando-se as restrições de precedência do projeto. Em conseqüência,

$$li(0) = vpl(GIL, 1) + vpl(PdS, 9) + vpl(PsS, 9) + vpl(Pc, 9) + vpl(CD, 9) +$$

$$vpl(LP, 9) + vpl(SC, 9) + vpl(CP, 9) + vpl(CLM, 9)$$

$$= -49 + 15 + 31 + 11 - 10 - 5 - 71 - 6 + 441 = 357$$

Passo 2: *Efetuando o branch inicial*

A busca da seqüência de unidades de software que maximiza o valor do projeto prossegue com a inserção na árvore dos nós correspondentes às unidades de *software* que podem ser construídas em seguida. Neste caso, a escolha é obvia, dado que a árvore possui apenas o nó zero e a unidade *GIL* é a única que pode ser construída no momento. A Figura 3 mostra o estado da árvore após a inserção do nó correspondente à unidade *GIL*. O cálculo do limite superior e inferior do nó 1, seguem os procedimentos descritos no Passo 1.

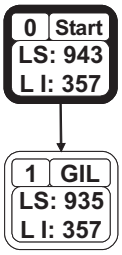

Figure 3. Primeira expansão de nó da árvore de busca

Passo 6: *Escolhendo o nó para o branch*

A busca prossegue de acordo com os passos já descritos até que a árvore atinja o estado descrito na Figura 4, quando uma escolha têm que ser feita:

Qual nó deverá ser expandido?

Neste ponto o algoritmo utiliza a heurística do nó com o maior potencial, selecionando o nó que apresenta o maior limite superior para ser expandido, isto é o nó 4.

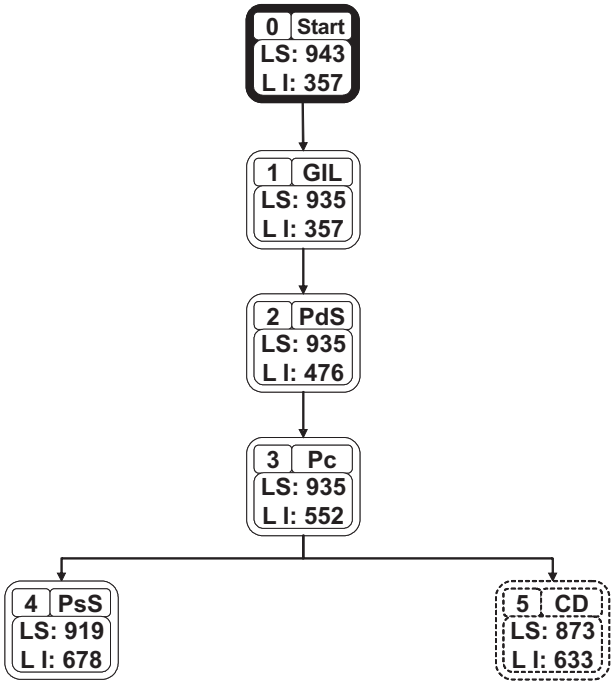

Figure 4. Quarta expansão de nó da árvore de busca

Passo 13: *Identificando a solução ótima*

A busca continua até que a árvore atinja o estado descrito na Figura 5. Neste instante, não existe nenhum nó que ofereça uma solução melhor do que a do nó 18, que não pode ser expandido. Em conseqüência, a solução ótima tem um VPL de $878 e corresponde à seqüencia

$$GIL \rightarrow PdS \rightarrow Pc \rightarrow PsS \rightarrow SC \rightarrow CD \rightarrow CP \rightarrow LP \rightarrow CLM$$

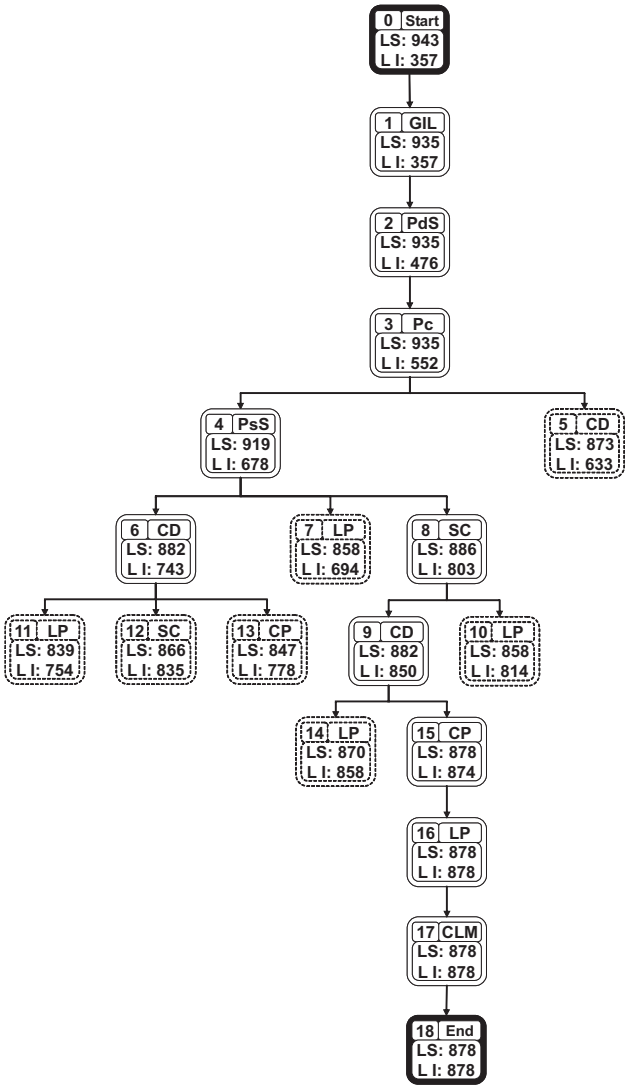

Figure 5. A árvore de busca gerada pelo método branch & bound

É importante se observar que nem todos os nós da árvore são expandidos durante o processo de busca pela solução ótima. Por exemplo, o nó 5 jamais é expandido. Na verdade, assim que o nó 15 é inserido na árvore, o nó 5 deixa de ser considerado para expansão. Isso ocorre porque o seu limite superior (que indica o maior valor que uma seqüencia de unidades de *software* que inicie com $GIL \rightarrow PdS \rightarrow Pc \rightarrow CD$ pode atingir) é menor do que o limite inferior do nó 15 (que indica o menor valor que uma seqüencia que inicie com $GIL \rightarrow Pds \rightarrow Pc \rightarrow PsS \rightarrow SC \rightarrow CD \rightarrow CP$ pode propiciar).

4. O Método Branch & Bound

Em termos formais, o algoritmo *branch & bound* para a maximização do VPL de projetos de *software* é descrito da seguinte forma. Dado

- Um grafo de precedências de unidades de *software* $G(V_G, E_G)$, composto pelo conjunto de unidades de *software* V_G, e pelo conjunto das restrições de precedência entre as unidades E_G;
- Uma janela de oportunidade P e
- Uma taxa de desconto r.

A seqüência S de unidades de software $v_i \in V_G$, que atende às restrições de precedência descritas em E_G e tal que *vpl*(S) é máximo, é dada por $B\&B(G, P, r)$, onde:

$$B\&B(G,P,r) \leftarrow \begin{cases} \Omega_T \leftarrow \{Start\}; \Theta_T \leftarrow \emptyset; Q \leftarrow \{Start\}; \\ \text{Repita:} \\ \quad N \leftarrow q \in Q, \text{ tal que } ls(q) = \textit{Máximo}(\{ls(q')|q' \in Q\}), \\ \quad \Omega_T \leftarrow \Omega_T \cup eligible(N), \\ \quad \Theta_T \leftarrow \Theta_T \cup \{(N,e)|e \in eligible(N)\}, \\ \quad \textit{MaiorLI} \leftarrow \textit{Máximo}(\{li(v)|v \in \Omega_T\}), \\ \quad Q \leftarrow \{v \in \Omega_T | v \in (Q - \{N\}) \cup eligible(N) \wedge ls(v) \geq \textit{MaiorLI}\}, \\ \text{Até que } Q = \emptyset; \\ S \leftarrow pathTo(N). \end{cases}$$

onde:

- S é a melhor solução encontrada entre os candidatos em Q;
- Ω_T é o conjunto de nós da árvore de busca;
- Θ_T é p conjunto de arestas da árvore de busca;
- *MaiorLI* é o maior limite inferior encontrado;
- Q é a lista de nós candidatos.

4.1. A Heurística do Limite Superior

A função do limite superior *ls*$(N \in \Omega_T) \rightarrow \mathbb{R}$ avalia o VPL máximo que uma solução pode ter, considerando a seqüência parcial construída até o nó N.

$$ls(N) \leftarrow \sum vpl(s_i, i) \text{ tal que } s_i \in pathTo(N) + \\ \sum vplMax(v_j, when(N, v_j)) \text{ tal que } v_j \in (V_G - \{v_k | v_k \in pathTo(N)\})$$

4.2. A Heurística do Limite Inferior

A função de limite inferior *li*$(N \in \Omega_T) \rightarrow \mathbb{R}$ avalia o VPL mínimo do subespaço de busca do nó candidato N, considerando a seqüência parcial conhecida.

$$li(N) \leftarrow \sum vpl(s_i, i) \text{ tal que } s_i \in pathTo(N) + \\ \sum vplMin(v_j, when(N, v_j)) \text{ tal que } v_j \in (V_G - \{v_k | v_k \in pathTo(N)\})$$

4.3. Funções Auxiliares

O método *branch & bound* também faz uso das seguintes funções:

- A função *eligible*$(N \in \Omega_T) \rightarrow V_G$ que informa o conjunto de nós que estão habilitados para serem os sucessores imediatos de um nó N na árvore de busca.
- A função *pathTo*$(N \in \Omega_T) \rightarrow$ Seq V_G que informa a seqüência de unidades de *software* que leva até o nó N da árvore.
- A função *when*$(v_i \in V_G, N \in \Omega_T) \rightarrow P$ que informa o período mais cedo em que uma unidade v_i pode ser desenvolvida, considerando a seqüência parcial até *N*.

5. Discussão

Abaixo encontram-se as respostas para questões chaves sobre as implicações do desenvolvimento de um método *branch & bound* para encontrar a seqüencia de unidades de *software* que maximiza o VPL de um projeto de *software*.

5.1. Por que utilizar um método *branch & bound* para maximizar o VPL?

A quantidade de seqüências possíveis para o desenvolvimento das unidades de um projeto de *software* tende a crescer exponencialmente com o número de unidades em que o projeto está dividido. Conseqüentemente, só é viável enumerar todas estas seqüências quando o projeto está dividido em poucas unidades. Isto faz com que o problema de encontrar a seqüência que maximiza o VPL do projeto se beneficie da existência de métodos heurísticos como o *branch & bound*, que, na grande maioria dos casos, não precisa enumerar todas as seqüências para encontrar a solução ótima [Liberti 2003].

5.2. O que este método tem de melhor se comparado ao IFM?

Existem duas grandes vantagens na utilização do método *branch & bound* em substituição ao IFM. A primeira vantagem é que este método garante uma solução ótima para o problema, enquanto que o IFM se contenta com resultados inferiores. Em projetos que custem milhões de dólares americanos, uma diferença de cerca de 10% da solução propiciada pelo IFM para a solução ótima significa a perda de centenas de milhares de dólares. Desperdícios desta magnitude pode levar a perda de competitividade, favorecendo o crescimento da concorrência. A segunda vantagem é que o método pode ser aplicado à projetos que apresentem relações de dependência múltiplas entre as unidades de *software*. Embora esta seja uma situação comum no universo dos projetos de *software*, tal construção não é suportada pelo IFM.

5.3. Qual o impacto esperado nas negociações das propostas de desenvolvimento?

Diante da competição cada vez mais acirrada por espaço no mercado de bens e serviços, nos dias de hoje, muitas empresas terceirizam o desenvolvimento de *software*, em todo ou em parte, estabelecendo um ambiente competitivo saudável entre as empresas prestadoras de serviços, que passaram a competir entre si por contratos de desenvolvimento de *software*. Propostas que maximizem o retorno dos investimentos a serem efetuados pelos clientes tem uma vantagem competitiva óbvia sobre aquelas que adotam uma visão mais conservadora do desenvolvimento de *software*. Nestas circunstâncias o método *branch & bound* proposto neste artigo sempre identifica a solução ótima, oferecendo o melhor retorno de investimento possível às empresas contratantes de serviço de desenvolvimento de *software*.

6. Conclusão

Este artigo apresentou um método *branch & bound* que identifica um plano de desenvolvimento que maximize o valor de um projeto de *software* para as organizações. O método sempre encontra a solução ótima e não impõe restrições de precedência irrealistas entre as unidades de *software*, favorecendo o desenvolvimento de *softwares* complexos a partir de um investimento inicial reduzido.

Referências

Abacus, A., Barker, M., and Freedman, P. (2005). Using test-driven software development tools. *Software, IEEE*, 22(2):88–91.

Denne, M. and Cleland-Huang, J. (2004a). The incremental funding method: data-driven software development. *IEEE Software*, 21(3):39–47.

Denne, M. and Cleland-Huang, J. (2004b). *Software by Numbers - Low-Risk, High-Return Development*. Prentice Hall.

Denne, M. and Cleland-Huang, J. (2005). Financially informed requirements prioritization. In Roman, G.-C., Griswold, W., and Nuseibeh, B., editors, 27^{th} *international conference on Software Engineering*, pages 710–711, St. Louis, MO, USA. ACM.

Fabozzi, F. J., Davis, H. A., and Choudhry, M. (2006). *Introduction to Structured Finance*. John Wiley.

Gross, J. L. and Yellen, J. (2005). *Graph Theory and Its Applications*. Chapman & Hall and CRC, 2^{nd} edition.

Helo, P., Hilmola, O.-P., and Maunuksela, A. (2004). Managing the productivity of product development: a system dynamics analysis. *International Journal of Management and Enterprise Development*, 1(4):333–344.

Highsmith, J. (2002). *Agile Software Development Ecosystems*. Addison-Wesley.

Hillier, F. S. and Lieberman, G. J. (2001). *Introduction to operations research*. McGraw-Hill, New York, NY, 7^{th} edition.

Jorgenson, D. W., Ho, M. S., and Stiroh, K. J. (2003). Growth of us industries and investments in information technology and higher education. *Economic Systems Research*, 15(3):279–325.

Lam, H. (2004). New design-to-test software strategies accelerate time-to-market. In Goetz, M., editor, 29^{th} *International Electronics Manufacturing Technology Symposium*, pages 140–143, San Jose, CA, USA. IEEE.

Liberti, L. (2003). *Optimization and Optimal Control*, chapter Comparison of Convex Relaxations for Monomials of Odd Degree, pages 165–174. Computers and Operations Research. World Scientific Publishing Company, Hackensack, NJ.

McManus, J. C. (2003). *Risk Management in Software Development Projects*. Elsevier.

Nord, R. and Tomayko, J. (2006). Software architecture-centric methods and agile development. *Software, IEEE*, 23(2):47–53.

Steindl, C. (2005). From agile software development to agile businesses. In Matos, J. S. and Crnkovic, I., editors, 31^{st} *EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 258– 265, Porto, Portugal. Porto University.

Whittle, R. and Myrick, C. B. (2005). *Enterprise Business Architecture*. Auerbach.