Relatorio - Aplicação do GRASP

Giovanne da Silva Santos

12 de Janeiro de 2021

Resumo

O gás lift é uma das técnicas artificiais mais comum de injeção para a produção de óleo. E isto está vinculado com o aumento de velocidade e precisão na injeção de gás neste petróleo. Contudo, a aplicação demasiada de gás em certo poços, prejudicará na produção de óleo por conta da pressão e atrito que o gás causará nos dutos em que o óleo percorre, assim como o desperdício de gás que afetará o produtor de petróleo economicamente. Além disso, caso não aplique-se gás suficiente, então não haverá pressão suficiente para ejetar o óleo presente no poço. Por conta disso, a resolução do problema de otimização do gás lift é essencial na produção de petróleo a fim de minimizar os custos e maximizar a produção.

1 Introdução

Caso um determinado reservatório de petróleo não consegue emergir os fluídos até a superfície, então uma técnica de injeção faz-se necessário, mas tal técnica terá de ser aplicada da melhor forma e otimizada possível para fins econômicos. Uma técnica artificial de injeção reduz a pressão da coluna do fluido causando, assim, uma redução da pressão do fundo poço. Por conta disso, uma grande diferença de pressão entre o reservatório e o fundo do poço é criada, e, consequentemente, os fluidos irão emergir até a superfície. O processo de gás lift é umas das técnicas artificiais de injeção mais utilizadas. No gás lift, o gás é injetado na parte inferior da tubulação, e o óleo é produzido através da impulsão feita pela expansão do gás e na redução de pressão hidrostática da coluna do fluido que está dentro do poço.

2 Descrição dos problemas

No problema de otimização de gás lift, temos dois cenários. O primeiro cenário consiste em maximizar a produção de óleo dado uma quantidade suficiente disponível de gás para injeção, e o segundo cenário tem o objetivo de minimizar a quantidade de gás dado uma quantidade de óleo produzido pré-planejada. A otimização, de um modo geral, leva em conta com a curva de performance do gás lift e fatores econômicos.

Para o cenário em que queremos maximizar a produção de óleo e tendo uma quantidade suficiente de gás, temos a seguinte função objetiva:

$$max \sum_{1}^{n} Q_{o_i} = f(Q_{g_1}, Q_{g_2}, ..., Qg_i), i = 1, 2, ..., n$$

sendo que a quantidade total de gás injetada nos poços deverá ser igual ou menor que a quantidade de gás disponível conforme é representado nesta equação (Hamedi[4]):

$$\sum_{1}^{n} Q_{g_i} \leq A$$

onde Q_{o_i} é quantidade de óleo produzido por poço, n é a quantidade de poços, Q_{g_i} é quantidade de gás injetada em cada poço e A é a quantidade de gás disponível.

No segundo cenário, que é o cenário estudado neste trabalho, queremos minimizar quantidade de gás injetado e tendo uma quantidade de produção de petróleo já pré-determinada, teremos:

$$min \sum_{1}^{n} Q_{g_i}, i = 1, 2, ..., n$$

em que a quantidade total de petróleo será igual a quantidade já pré-determinada, como mostrado a seguir:

$$\sum_{1}^{n} Q_{o_i} = f(Q_{g_1}, Q_{g_2}, ..., Qg_i) = B$$

sendo B a quantidade pré-determinada.

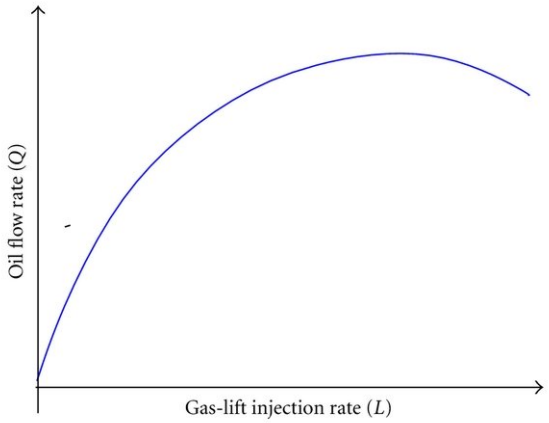
tendo também estas restrições em ambos os cenários:

$$Q_{g_i} \geq Q_{g_i min}$$

$$Q_{g_i} \leq Q_{g_i max}$$

Além disso, vale salientar que alguns poços produzem petróleo naturalmente ou que precisam de muito pouco gás injetado, assim como temos poços que não respondem bem à injeção de gás. Ademais, há os poços "mortos"no qual é necessária uma quantidade miníma de gás igual ou maior do que $Q_{g_i min}$ para poder produzir petróleo. Assim como a quantidade de gás não pode exceder em um poço, ou seja, ser maior do que $Q_{g_i max}$, a fim de não haver o decaimento da produção de petróleo por conta da percar de pressão que pode ocasionar na injeção do gás.

A quantidade máxima $Q_{g_i max}$ e miníma $Q_{g_i min}$ são obtidas através curva de performance do gás lift(GLPC) é ilustrado na imagem a seguir:



Em que temos no eixo y a produção de petróleo e no eixo x a quantidade de gás injetado. Como podemos ver, a quantidade ideal de gás injetado em um poço, deverá ser no pico da curva caso tenhamos uma quantidade de gás suficiente, senão, teremos que, ao menos, determina uma quantidade miníma de gás de tal modo que produza a maior quantidade possível ou pré-determinada de petróleo.

Para representar a curva GLPC, foi utilizado o modelo introduzido por [1] para gerar a equação que calcula a produção de petróleo a partir de uma gás injetado que é:

$$Q_o = a + b \times Q_g + c \times Q_g^2 + d \times \ln(Q_g + 1)$$

, em que a, b, c e d são os coeficientes que cada poço irá ter.

3 Fundamentação teórica

As técnicas implementadas foram a busca local, path-relinking e o GRASP. Nesta seção, será dado uma explicação básica e geral dessas técnicas.

3.1 Busca local

A busca local é um método simples,porém poderoso. Ele consiste em encontrar uma solução a partir de uma vizinhança de uma solução qualquer(um candidato) a fim de achar uma solução melhor - o que passa a ser o novo candidato. Aplica-se esse principio até não ter como melhorar a solução localmente.

Segue o pseudo-algoritmo desenvolvido para a busca local:

Algorithm 1 Busca Local

```
1:  procedure BUSCA LOCAL(s_0)
2:      best ← s_0
3:      candidate ← BESTNEIGHBOR(s_0)
4:      repeat
5:          if candidate < best then
6:              best ← candidate
7:          end if
8:          candidate ← BESTNEIGHBOR(canditate)
9:      until Não há mais vizinhos melhores do que best
10:     return best
11: end procedure
```

O algoritmo inicializa, na linha 2, temos a atribuição da melhor solução atual com a solução inicial recebída como parâmetro. Na linha 4 temos o BESTNEIGHBOR que gera as vizinhanças a partir de uma técnica de modificação na qual altera cada elemento da solução. Ao entrar no laço, na linha 5, temos: o percorrimento pela vizinhança investigando se o candidato (*candidate*) tem menor custo do que a melhor solução (*best*) na linha 6; na linha 6, caso seja a melhor possível, então *candidate* atribuir o valor à solução *best* ; na linha 8, é investigado novamente uma nova vizinhança. Na linha 9, temos a finalização do laço quando não tivermos mais como achar vizinhos melhores do que *best*. Na linha 11, temos o retorno da solução *best*.

3.2 Path-relinking

O path-relinking é uma técnica que gera soluções baseadas em duas soluções, uma sendo inicial e outra final. De modo geral, a partir da inicial ao mover um valor, que na solução final está em outra posição, para a posição na qual o mesmo valor está na solução final, assim criando uma nova solução e analisando qual é a melhor. Novas soluções serão geradas até que não tenha mais como mover os valores, ou seja, a inicial seja igual a final, ou até que certa porcentagem arbitrária da inicial seja modificada.

Considere a criação de caminhos que unem duas soluções selecionadas x' e x'' produzindo uma solução $x' = x(1), x(2), x(3), ..., x(k) = x''$. Com a finalidade de diminuir a quantidade de opções, a solução $x(i+1)$ deve ser criada a partir de $x(i)$ e a cada passo escolhe-se um movimento que reduza a quantidade de passos para chegar a x'' [3].

Vale ressaltar que, para o path-relinking, é necessário uma diferença simétrica de 4 componentes entre as soluções ou como denominada de distância *Hamming* para que haja um mínimo local entre elas. Isso é descrito e demonstrado em [6].

O algoritmo a seguir é um pseudo-algoritmo que representa o path-relinking (do tipo *forward relinking*):

Algorithm 2 Path Relinking

```
1: procedure PATH RELINKING(SoluçãoInicial, SoluçãoFinal)
2:     Δ ← DistânciaSimetrica(SolucaoInicial, SolucaoFinal)
3:     best ← MelhorSolução(SolucaoInicial,SolucaoFinal)
4:     while Δ ≠ 0 do
5:         Movimento(SolucaoInicial)
6:         best ← MelhorSolucao(best,SolucaoInicial)
7:     end while
8: end procedure
```

Nesse pseudo-algoritmo, temos na linha 2 o cálculo da diferença simétrica entre as duas soluções passadas como parâmetro para saber quantas soluções serão geradas e testadas e depois escolhe a melhor solução entre as duas para guardar em *best*. Em seguida, dentro do laço, é usado a função *Movimento* que modifica a solução inicial baseada na solução final. Na linha 6, é atualizado a melhor solução. No caso, esse pseudo-código representa um simples path-relinking que não se preocupa em começar com a melhor ou pior solução como a forward ou backward. Entretanto será experimentado o path-relinking *mixed* no qual tanto a solução inicial quanto a final movem-se até uma distância equidistante.

3.3 GRASP

O GRASP é uma técnica que constrói uma solução a partir de uma lista restrita de candidatos(LRC). A proposta desta técnica é não ser totalmente gulosa e nem totalmente aleatória. Ao construir uma solução depois de um procedimento tanto guloso quanto aleatório e adaptativo, o GRASP aplica ,nesta solução, uma investigação de soluções semelhantes a a solução construída(vizinhança) com uma abordagem multi-start até não ter como melhorar a solução.

A seguir, um pseudo-algoritmo:

Algorithm 3 GRASP

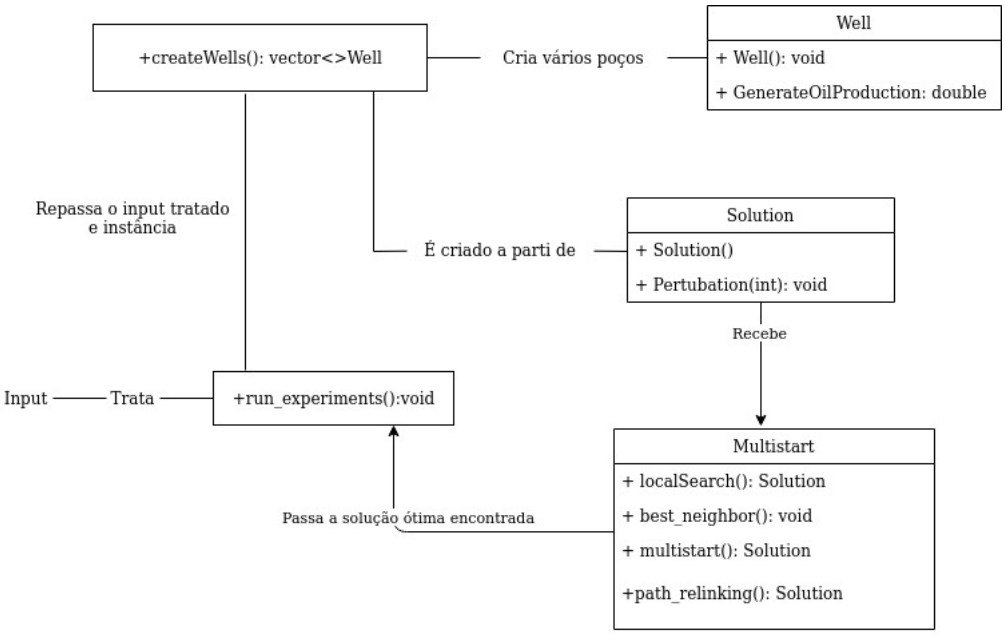
```
1: procedure GRASP
2:     while Regra de parada não for verdadeira do
3:         s_0 ← {∅}
4:         while Uma solução não é construída do
5:             Selecione um conjunto de variáveis LRC
6:             Selecione aleatoriamente um elemento s ∈ LRC
7:             Faça s_0 ← s_0 ∪ {s}
8:             Atualize os critérios para LRC com a inclusão {s}
9:         end while
10:    end while
11:    while s = 0 do
12:        Determine a melhor solução t ∈ N(s_0)
13:        if f(t) < f(s_0) then
14:            s_0 ← t e s ← 1
15:        else
16:            s ← 0
17:        end if
18:    end while
19:    return s_0
20: end procedure
```

No segundo laço, temos a escolha de uma lista restrita, depois é escolhido aleatoriamente um elemento dessa lista, e, depois da escolha, esse novo componente é colocando na solução. O laço termina quanto já for terminado a construção da solução. Em seguida, no segundo laço, é investigado a vizinhança da solução construída, se acharmos um vizinho melhor, então ele é escolhido e repete-se a investigação. O procedimento terminará quando não acharmos mais vizinhos melhores.

4 Descrição do algoritmo implementado

Neste seção será explicado as classes e funções mais relevantes para resolução do problema utilizando as técnicas explicadas na seção passada.

Primeiramente será mostrado um diagrama que ilustra o caso de uso do algoritmo.



O input recebido - que é um txt contendo, na primeira linha, a quantidade de poços, e em seguida uma sequência de linha com os valores de gás injetado e outra linha com petróleo produzido baseado nos valores da linha do gás - é convertido pela função run experiments em uma matriz de gás injetado e outra de petróleo produzido baseado em cada elemento da matriz de gás, e que cada linha dessas matrizes representam um poço. Na mesma função, temos a instância e repasse dessas matrizes à função createWells que, a partir do construtor de Well, mapeia as matrizes no respectivo poço. Na classe Well, temos o construtor que a parti de um vetor gás e um vetor petróleo, calcula os coeficientes daquele poço com base em um método de resolução de sistema linear e, também, há o cálculo do petróleo produzido a partir do gás injetado.

A classe solução utiliza o construtor para rotular a prioridade do poço com base na quantidade de petróleo produzido a partir da menor quantidade de gás injetado, ou seja, o poço com maior prioridade, no qual é 1, será aquele que mair produz utilizando a menor quantidade gás. Além disso, neste construtor, há a construção da solução de forma gulosa e aleatória como ilustrado no pseudo-algoritmo a seguir:

Algorithm 4 Solution

```
1: procedure SOLUTION(vector¡Wells¿wells,número de poços)
2:     s_0 ← {∅}
3:     while Faça até a solução ser construída do
4:         a ← wells.tamanho()*(prioridade/10) - 0.1
5:         b ← wells.tamanho()*(prioridade/10)
6:         Selecione um conjunto LRC de valores entre a e b
7:         Selecione aleatoriamente um elemento s ∈ LRC
8:         Faça s_0 ← s_0 ∪ {s}
9:     end while
10:    return s_0
11: end procedure
```

A construção da solução será O(n), tendo em vista que n é o tamanho do vetor de solução e não precisará fazer nenhuma outro laço além do da construção. No caso, a proposta de escolha desse LRC é pegar os potenciais menores valores que podem produzir naquele poço sem precisa está no pico da curva glpc. Em seguida temos a função pertubation na mesma classe que será necessária para construção da vizinhança:

Algorithm 5 Pertubation

```
1: procedure PERTUBATION(índice) Altere o valor da solução
2:     Selecione aleatoriamente um valor a entre 0.75 e 1.5
3:     elemento ← solução[índice]
4:     elemento ← elemento * a
5:     Atualize a produção de petróleo e a quantidade de gás injetado desta
   solução
6: end procedure
```

No pertubation, a complexidade será O(1), pois já saberemos o índice do valor que deverá ser alterado. Em seguida, temos a classe multistart que terá todas a técnicas apresentadas e descritas de forma geral na seção passada. Primeiramente teremos a função best_neighbor que funcionará na seguinte forma:

Algorithm 6 best_neighbor

```
1: procedure BEST_NEIGHBOR(s_0)
2:     neighbor ← s_0
3:     checkpoint ← neighbor
4:     i ← 0
5:     repeat
6:         neighbor.pertubation(i)
7:         if neighbor < best  then
8:             s_0 ← neighbor
9:         end if
10:        neighbor ← checkpoint
11:    until i == neighhbor.tamanho()
12: end procedure
```

Já neste algortimos, teremos uma complexidade de O(n), em que n é o tamanho do vetor neighbor, e em cada índice utilizaremos o pertubation que é O(1). Logo, a complexidade de best_neighbor será O(n). A partir de uma solução é construído uma vizinhança que é extraído o melhor vizinho caso haja. Vale ressaltar que o comparador ≪"checa se o elemento do lado esquerdo tem a quantidade de gás injetado menor do que o lado direito e também se a produção de petróleo é maior ou igual à quantidade de petróleo pré-planejado. Agora, teremos uma função da busca local que é representado como localSearch. Ilustra-se a seguir no pseudo-código:

Algorithm 7 localSearch

```
1: procedure LOCALSEARCH(s_0)
2:     best ← s_0
3:     candidate ← BEST_NEIGHBOR(s_0)
4:     improve ← true
5:     while improve do
6:         if candidate < best then
7:             best ← candidate
8:         end if
9:         if best < candidate then
10:            improve ← false
11:        end if
12:        candidate ← BEST_NEIGHBOR(canditate)
13:        return best
14:
```

A lógica é a mesma da busca local geral, a diferença seria apenas no critério de comparação já explicado em que a representação de "não ter mais como melhorar"está na mudança da variável improve que passará a ser false caso best seja melhor do que candidate. Nesta técnica, utilizamos o best_neighbor que é O(n) sendo n o tamanho do vetor solução e vezes o número de soluções melhores

que a atual encontrada. Por conta disso, nós teremos a complexidade de O(n× Nº de soluções melhores encontradas).

O path-relinking escolhido foi o mixed, no qual a solução inicial e guia andam juntas até que estejam semelhantes. Ela utilizará uma função auxiliar chamada move que funciona da seguinte forma:

Algorithm 8 Move

```
1: procedure MOVE(SolInicial,SolGuia,índice)
2:     auxiliar ← SolInicial[índice]
3:     SolInicial[[indice] ← SolGuia[índice]
4:     SolGuia[índice] ← auxiliar
5: end procedure=0
```

A complexidade do move será O(1) já temos o índice que deverá ser alterado em cada solução. No caso, será apenas uma espécie de troca de elementos entre as duas soluções. A partir disso, teremos o funcionamento do path-relinking que será:

Algorithm 9 Path-Relinking

```
1: procedure PATH-RELINKING(SolInicial,SolGuia)
2:     Calcular diferença entre SolInicial e SolGuia
3:     if A diferença for menor do que 4 then
4:         Retorne a melhor dentre as duas
5:     end if
6:     best ← A melhor dentre as duas
7:     i ← 0
8:     while Faça até que as duas sejam iguais do
9:         Move(SolInicial, SolGuia, i)
10:        Atualize best escolhendo a melhor dentre as duas
11:        i + = 1
12:    end while
13:    Retorne best
14: end procedure
```

Semelhante ao path-relinking já explicado, a diferença é, por ser mixed em vez de forward, a solução inicial e a guia serão modificadas até que elas sejam semelhantes e sempre comparando a melhor dentre as duas. Como está sendo utilizado o mixed, então será alterado as duas soluções simultaneamente, então a complexidade será O(n), tendo em vista que será percorrido nas duas soluções ao mesmo tempo. Por fim, foi implementado o GRASP que será:

Algorithm 10 GRASP

```
1: procedure GRASP(wells)
2:     Construa com s_0 ← Solution(wells)
3:     best ← s_0
4:     improve ← true
5:     candidate ← s_0
6:     SolInicial ← s_0
7:     while improve do
8:         candidate ← localSearch(candidate)
9:         SolInicial ← candidate
10:        if best < candidate then
11:            improve ← false
12:        else
13:            best ← candidate
14:        end if
15:    end while
16:    Retorne Path-Relinking(SolInicial,best)
17: end procedure
```

Como mostrado, será construído a solução com a lógica explicada do construtor de solução, e será aplicado a busca local até não ter achado uma solução melhor do que best. Por fim, será aplicado o path-relinking entre a melhor solução e a segunda melhor solução encontrada. Como o construtor tem complexidade O(n), busca local de O(n× Nº de soluções melhores encontradas) e path-relinking de O(n), então teremos que a complexidade total será de (n + (n× Nº de soluções melhores encontradas + n), ou seja, O(2n + (n Nº de soluções melhores encontradas).

5 Descrição dos experimentos computacionais

As instâncias utilizadas foram retiradas no Buitrago et al[2], lá temos duas tabelas, um com 6 poços e outras com 56 poços. Em run_experiments, é feito a leitura em do txt que tem, na primeira linha o número de poços, e a sequência, em seguida, da linha de gás injetado em u poço e na próxima linha o petróleo produzido no poço com base nos valores do gás injetado na linha anterior e assim sucessivamente com cada dupla de linha representando um poço. A saída do programa grava em um arquivo e imprime no termina o número de poços da instância, quantidade de petróleo pré-planejada, o vetor da solução gerada, petróleo produzido, a quantidade mínima e a máxima de petróleo possível e o gás injetado a partir da solução encontrada. A partir dessas instâncias, utiliza-se um método gaussiano para calcular os coeficientes de cada poço que irão calcular a quantidade de petróleo baseado no gás injetado no poço.

O ambiente de teste foi um notebook Lenovo com processador Intel Core i7-7500U CPU 2.70GHz x 4, memória RAM de 3,6 GB e HD de 500 GB e sendo

o sistema operacional o Ubuntu 20.0.

Para a compilação do programa, através do terminal, basta digitar "make"na pasta do programa. Para execução do programa, será necessário entrar na pasta "build"e digitar ./exe.out [nome da instâncias]. Até o momento, temos as instâncias well-six.txt e well-fifty-six.txt.

6 Resultados obtidos: análise e discussão

Nesta seção será analisado os experimentos através de tabelas e, para efeitos de comparação, será mostrado a quantidade de gás injetado pela melhor solução encontrada e a produção de petróleo produzido

Primeiro, será apresentado os resultado com 6 poços que terá uma produção mínima de petróleo de 1772.9, utilizando 1587 de gás, e máxima de 4172.3, logo:

Qtd de petróleo pré-planejada	Petróleo produzido	Gás injetado
2086.15	2643.95	2692.21
2172.3	2790.794	3294.15
1772.9	1785.91	2286.13

Tabela 1: Resultados com 6 poços

Já para os resultados da instância com 56 poços que tem quantidade mínima de produção de petróleo de 17271 com injeção de gás de 23425 e máxima de 23742:

Qtd de petróleo pré-planejada	Petróleo produzido	Gás injetado
13306	21822	105540
24612	21817.6	105548
17271	21811.8	105559

Tabela 2: Resultados com 56 poços

Na maioria dos resultados é mostrado que as melhores soluções encontradas ultrapassaram a quantidade de petróleo pré-planejada mesmo quando utilizamos a quantidade mínima de petróleo como quantidade pré-planejada, temos uma quantidade acima da quantidade de gás mínima. Com 6 poços, tivemos o encontro mais aproximado de petróleo em relação à quantidade pré-planejada. Em 56 poços a diferença aumenta e evidenciando os casos em que acaba preso em ótimos locais, pois é achado uma solução de menor injeção de gás porém não consegue achar outras ao ponto de ter pouca diferença da quantidade de gás mínima.

7 Conclusão

Os problemas de otimização associados ao gás lift são bastante estudados, principalmente, por conta do valor econômico que traz com a resolução de tais problemas. Por conta disso, temos várias meta-heurísticas e modelagens aplicadas a fim de trazer melhores resultados. Contudo, o segundo cenário ainda não tem muitos estudos envolvendo ele, tornando-se necessário criar e aplicar novos algoritmos para a resolução como Namdar[5] mencionou.

Com os uso do algoritmo de multi-start e com os parâmetros implementados, ainda tem-se riscos de cairmos em ótimos locais assim como a modelagem do problema deve ser melhorada pois há riscos dos coeficiente calculados por poços ainda não estar muito precisos. Por conta disso, novos parâmetros e refinamento das técnicas e modelagem devem ser necessárias para termos mais êxito na resolução da minimização de injeção de gás.

Referências

[1] Gabriel A Alarco´ n, Carlos F Torres, and Luis E Go´ mez. Global optimization of gas allocation to a group of wells in artificial lift using nonlinear constrained programming. *J. Energy Resour. Technol.*, 124(4):262–268, 2002.

[2] S Buitrago, E Rodriguez, D Espin, et al. Global optimization techniques in gas allocation for continuous flow gas lift systems. In *SPE gas technology symposium*. Society of Petroleum Engineers, 1996.

[3] Fred Glover, Manuel Laguna, and Rafael Martí. Fundamentals of scatter search and path relinking. *Control and cybernetics*, 29(3):653–684, 2000.

[4] H Hamedi, F Rashidi, and E Khamehchi. A novel approach to the gas-lift allocation optimization problem. *Petroleum Science and Technology*, 29(4):418–427, 2011.

[5] Hamed Namdar. Developing an improved approach to solving a new gas lift optimization problem. *Journal of Petroleum Exploration and Production Technology*, 9(4):2965–2978, 2019.

[6] Mauricio GC Ribeiro, Celso C e Resende. Path-relinking intensification methods for stochastic local search algorithms. *Journal of heuristics*, 18(2):193–214, 2012.