Iterated Local Search
(Algortimo de busca local iterada)

Isaac Kosloski Oliveira[1]

[1]Faculdade de Computação
Universidade Federal de Mato Grosso do Sul (UFMS) Campo Grande – MS – Brazil

Abstract. *Baseado no artigo Iterated Local Search [?]*

1. Introdução

→ Algoritmos de alta performance para **problemas difíceis de otimização** não podem ser discretos;
→ Métodos para solução: **metahuerísticas**;
→ A **modelagem da metahuerística** deve ter um conceito e implementação simples;
→ A metaheurística deve ser eficiente e, se possível, de propósito geral;
→ O caso ideal é quando uma metaheurística pode ser usada sem nenhuma dependência de conhecimento do problema;
→ As metaheurística tem se tornado mais sofisticadas;
→ O caso ideal tem sido posto em segundo plano, na busca por maior desempenho;
→ Como consequência, conceitos específicos do problema devem ser incorporados na metaheurística, afim de de atingir o ”nível de arte”;
→ O limite entre heurítisca e metaherística se torna difuso e corre o risco de perder a simplicidade no conceito e implementação;
→ Para contrapor isto, aborda-se a modularidade e tenta decompor o algortimo da metaheurística em partes com especificidade distintas;
→ Deseja-se uma parte totalmente de propósito geral e uma outra somente para implementar o problema específico;
→ Pretende-se deixar a heurística embarcada sem alterações, por sua potencial complexidade;
→ Assim, tratar a heurística como uma ”caixa-preta”;
→ O ILS provê uma maneira simples de atingir essas especificações;
→ A essência da metaheurística ILS pode ser dada em uma ”*nut-shell*”:
  - Iterativamente, é gerado uma sequência de soluções pela heurística embarcada;
  - Soluções melhores comparadas com repetidas soluções geradas aleatóriamente pela heurística;
→ A ideia é simples e descoberta por vários autores;
→ Levou a ideia receber vários nomes diferentes, tais como:
  - iterated descent;
  - large-step Markov chains;
  - iterated Lin-Kernighan;
  - chained local optimization;
  - Combinações destes;
→ Existem dois pontos principais que caracterizam um algoritmo como uma ILS:

(i) Deve seguir-se apenas uma ”corrente” em que faz-se a busca (são exlcuídos algoritmos baseados em população);

(i) A busca por soluções melhores ocorre em um espaço reduzido definido pela saída da heurística ”caixa-preta”;

→ Na prática, a busca local é a heurística embarcada mais frequêntemente utilizada;
→ De fato, pode-se utilizar otimizadores, seja determinístico ou não;
→ Apesar de sua simplicidade conceitual, leva a resultados de ”nível artístico” sem a necessidade de usar muitos conceitos específicos do problema;
→ Isso pode acontecer, devido à maleabilidade do ILS, que permite muitas opções de implementação ao desenvolvedor;

2. Iterando uma busca local

2.1. Framework geral

→ Assume-se que, dado um algoritmo de otimização de heurística de problema específico, refere-se como *busca local*;
→ Esse algoritmo é implementado como uma rotina chamada $LocalSearch$;
→ A pergunta então é: ”pode-se otimizar tal algoritmo utilizando iteração?”. A resposta é: ”SIM”;
→ A otimização obtida de fato é significante;
→ Somente em alguns casos onde o método de iteração é incompatível com a busca local a otimização será mínima;
→ Assim, para obter uma melhor otimização possível, é necessário um entendimento do modo como o $LocalSearch$ funciona;
→ Seja C a função custo do nosso problema de otimização combinatória, C deve ser *minimizado*;
→ Nomeia-se soluções candidatas por s e S o conjunto de todas as soluçoes s;
→ Assume-se S como finito;
→ Assume-se que a *busca local* é deterministico e sem memória;
→ Mas na prática, muitas implementações do ILS possuem *busca local* não determinística e usam memória;
→ Dado uma entrada s, sempre terá a mesma saída $s*$, com custo menor ou igual ao $C(s)$;
→ O $LocalSearch$ então define um mapeamento de *muitos para um* do conjuto S para um conjunto menor $S*$ de soluções ótimas locais $s*$;
→ Para uma ilustração disso, introduz-se a ”bacia de atração” de um mínimo local $s*$ como o conjunto de s mapeado para $s*$ abaixo da rotina de busca local;
→ O $LocalSearch$ então leva uma solução inicial para uma solução no final da bacia de atração correspondente;
→ Agora, toma-se um s ou um $s*$ aleatoriamente;
→ Tipicamente, a distribuição de custos encontrados tem um crescimento rápido em partes de valores baixos;
→ A distribuição dos custos tem forma de sino, com média e variancia significantementes menores para soluções em $S*$ do que em S;
→ Como consequência, melhor usar busca local do que amostrar aleatoriamente em S, caso busca-se por soluções de baixo custo.
→ O elemento essencial para a busca local é uma estrutura de vizinhança;

→ Disso, S é um espaço com alguma estrutura topológica, e não somente um conjunto;
→ Dado tal espaço, permite-se mover de uma solução para outra de maneira inteligente;

2.2. Reinício aleatório

→ A possibilidade mais simples de melhorar acima de um custo encontrado pelo $LocalSearch$ é repetir a busca de um outro ponto de patida;
→ Todo $s*$ gerado é, então, independente;
→ Daí utilizar multiplos ensaios permite chegar na parte inferior da distribuição;
→ Essa abordagem é uma estratégia útil, principalmente quando as outras falham;
→ Essa abordagem falha à medida que as instancias crescem, uma vez que nesse limite a cauda de distribuição de custos colapsa;
→ Estudos empíricos e argumentos gerais indicam que algoritmos de busca local, em instâncias genéricas grandes, levam a custos que:
  (i) Tem média que é um excesso de porcentagem, fixa, abaixo do custo ótimo;
  (i) Tem uma distribuição que atinge um pico arbitrário em torno da média, quando a magnitude da instância tende ao infinito;
→ A segunda propriedade torna impossível, na prática, encontrar um $s*$ em que o custo seja um pouco menor, em termos percentuais, do que o custo típico;
→ No entanto, existem muitas soluções com custo menor significativo;
→ A amostragem aleatória tem uma probabilidade cada vez menor de encontrar soluções com custo menor à medida que o tamanho da instância aumenta;
→ Para alcançar essas configurações, uma amostragem guiada é necessária;
→ Isto é precisamente o que uma busca estocátisca realiza;

2.3. Pesquisando em $S*$

→ Afim de superar o problema associado com instâncias de tamanhos grandes é importante considerar o que a busca local faz:
→ Ela parte de um elemento de S onde C tem uma médica grande, para uma $S*$ onde C tem uma média menor;
→ É natural pensar nisso com recursão;
→ Usar a busca local afim de ir de $S*$ para um espaço menor $S**$, onde o custo será ainda menor;
→ Isso corresponde à uma busca local aninhada dentro de outra;
→ Tal construção pode ser iterado em quantos níveis desejar-se - uma hieraquia de buscas locais aninhadas;
→ Um exame mais minuscioso, demonstra que o problema é como formular uma buca local além do menor nível da hierarquia:
→ A busca local necessita de uma estrutura de vizinhança e isso não é dado à priori;
→ A dificuldade fundamental é definir uma vizinhança em $S*$ que pode ser enumerada e acessada eficientemente;
→ É desejável que o vizinho mais próximo em $S*$ seja relativamente próximo quando usado a distância em S;
→ Se esse não for o caso, uma busca estocástica em $S*$ tem uma pequena chance de ser eficiente;

→ Uma maneira de introduzir uma boa estrutura de vizinhança em $S*$:

1. Observe que uma estrutura de vizinhança em S induz uma estrutura de vizinha em um subconjunto de S;
2. Dois subconjuntos são vizinhos próximos simplemente se eles contem soluções que são vizinhos próximos;
3. Tome esses subjconjuntos afim de se serem as bacias de atração de $s*$;
4. Somos levados a identificar qualquer $s*$ com sua bacia de atração;

→ Isso nos dá a notação ”canônica” de vizinhança;

→ Podemos definir tal notação como:

- s_1* e s_2* são vizinhos em $S*$ caso as suas baciais de atração se ”toquem”;
- Isto é, possuem uma solução de vizinhos mais próximos em S;

→ Infelizmente, essa definição tem uma desvantagem de não poder, na prática, listar os vizinhos de $s*$, por não existir método computacional eficiente para encontrar todas as soluções de s na bacia de atração de $s*$;

→ Podemos gerar, estocasticamente, vizinhos mais próximos como:

- Partindo de $s*$, cria-se um caminho aleatório em S, s_1, s_2, ..., s_i, onde $s_j + 1$ é um vizinho mais próximo de s_j;
- Determine um s_j inicial, neste caminho, que pertença à uma bacia de atração distinta, então aplique a busca local para que s_j leve para um $s*' \neq s*$;
- $s*'$ é um vizinho mais próximo de $s*$.

→ Dado este procedimento, podemos performar uma busca local em $S*$;

→ Extendendo o argumento recursivamente, é possível observar a implementação de buscas aninhadas;

→ Realizando buscas locais em $S, S*, S**$, etc... de maneira hierárquica;

→ Infortunadamente, a implementação da busca do vizinho mais próximo no nível $S*$ tem um alto custo computacional;

→ Por consequência do grande número de vezes em que executa a $LocalSearch$;

→ É, assim, induzido o abandono da busca (estocástica) para os vizinhos mais próximos em $S*$;

→ Ao invés, usa-se uma uma noção de proximidade não tão acurada, permitindo uma busca estocástica rápida em $S*$;

→ Nossa construção leva à uma amostra (enviesada) de $S*$;

→ Tal amostra será melhor que uma aleatória;

→ Uma outra vantagem, dessa modificação de conceito de proximidade é a de não requerer definição de bacia de atração;

→ Assim, a busca local pode incorporar memória e ser não deterministica, fazendo com que o método seja de maior propósito geral;

2.4. Busca Local Iterada

→ Quer-se explorar $S*$ usando um caminho em que os paços de um $s*$ para um próximo, sejam sem a limitação de usar somente o vizinho mais próximo como definido previamente;

→ O ILS alcança isso heurísticamente, como segue:

- Dado um $s*$ atual, aplica-se, primeiro, uma perturbação que leva à um estado intermediário s' (pertencente ao S);

- A rotina $LocalSearch$ é aplicada para s', alcançando uma solução $s*'$ em $S*$;
- Se $s*'$ passa em teste de aceitação, isso se torna o próximo elemento do caminho em $S*$;
- Caso contrário, retorna para $s*$;

$\rightarrow$ O caminho resultante é um caso de busca estocátisca em $S*$, onde a vizinhança não é introduzida explicatamente;

$\rightarrow$ Este procedimento de busca local iterada deve levar à uma boa amostra enviesada, contanto que as perturbações não sejam, nem muito pequenas nem muito grandes;

$\rightarrow$ Se forem muito pequenas, irá recorrentemente voltar para $s*$ e poucas soluções novas de $S*$ será exploradas;

$\rightarrow$ Se forem muito grandes, s' será aleatório, não haverá viés na amostragem, assim, vamos recair em um algoritmo de reinício aleatório;

$\rightarrow$ O procedimento ILS geral é pictoricamente ilustrado na figura **??**;

$\rightarrow$ Para ser completo, assume-se que o caminho da busca local iterada não pode ser reversível;

$\rightarrow$ Isso não impede o ILS de ser muito efetivo na prática;

$\rightarrow$ Perturbações determinísticas podem levar à ciclos curtos (instâncias de for de tamanho 2);

$\rightarrow$ Assim, para evitar este tipo de cíclos, as perturbações devem ser aleatórias ou fazer com que elas sejam adaptativas;

$\rightarrow$ Caso a perturbação dependa de algum $s*$ anterior, ela possui o caminho em $S*$ com memória;

$\rightarrow$ Assim, tudo isso leva à definição do algoritmo ILS como metahuerística tendo a seguinte arquitetura de alto nível:

Algorithm 1 Iterated Local Search

```
PROCEDURE Iterated Local Search
    s0 = GenerateInitialSolution()
    s* = LocalSearch(s0)
    REPEAT
        s' = Perturbation(s*, history)
        s*' =LocalSearch(s')
        s* = AcceptanceCriterion(s*, s*', history)
    UNTIL termination condition met
END
```

$\rightarrow$ Na prática, muito da potencial complexidade do ILS está escondido da dependência do histórico;

$\rightarrow$ Caso aconteça de não existir tal dependência, então o caminho é sem memória:

$\rightarrow$ A perturbação e o critério de aceitação não dependem de de quaisquer soluções visitadas previamente durante o caminho, aceitando ou não $s*'$ com uma regra fixada;

$\rightarrow$ Isso leva à um caminho aletório dinâmico em $S*$ que é **"Markoviano"**;

$\rightarrow$ A probabilidade de performar um passo particular de s_1* para s_2*, depende somente destes;

→ A maior parte do trabalho de usar o ILS tem sido desse tipo;
→ Embora alguns estudos mostram inequivocamente que o uso de memória melhora a performânce;
→ O critério básico de aceitação, usa somente a diferença de custo de $s*$ e $s*'$;
→ Isto é similar ao que aconte no simulated annealing;
→ Um caso de limitação disto é aceitar somente movimentos de melhora, o que acontece com o simulated ennealing em temperatura zero;
→ O algoritmo realiza uma descida estocática em $S*$;
→ Se adicionado um método de critério de parada, por melhorias, baseado do tempo de CPU, resulta em um algortimo com duas buscas locais aninhadas;
→ Existe uma busca local operando em S embarcada em uma busca estocástica operando em $S*$;
→ Generalizando, pode-se extender esse tipo de algoritmo para mais níveis de aninhamento, possuindo uma busca estocástica distinta para $S*$, $S**$, etc...;
→ Cada nível é caracterizado por seu próprio método de perturbação e regra de parada;
→ Pode-se resumir esta seção por dizer que o potêncial poder da busca local iterada mente na sua amostragem enviesada do conjunto de ótimos locais;
→ A eficiência dessa amostragem depende tanto do tipode perturbação quanto do critério de aceitação;
→ Interessantemente, mesmo com as implementações mais ingênuas dessa partes, o ILS é melhor do que o reinício aleatório;
→ Ainda assim, muitas soluções melhores podem ser obtidas pela otimização dos módulos do ILS;
→ O critério de aceitação pode ser ajustado empiricamente, sem a necessidade saber-se nada sobre o problema sendo otimizado;
→ A rotina de perturbação pode incoporar quanta informação específica do problema quanto o desenvolvedor desejar;
→ Na prática a seguinte regra pode ser usada:
→ ”Uma boa perturbação transforma uma excelente solução em um excelente ponto de partida para a busca local”;
→ Juntos, esses aspectos demonstram que o ILS pode ter uma faixa larga de complexidade;
→ Mas a complexidade pode ser adicionada de maneira progrssíva e modularizada;
→ Isto permite o ILS ser uma metahuerística atraente, tanto para a academia quanto para a indústria;
→ A cereja do bolo é a velocidade:
→ Pode-se perfomar k buscas locais embarcada dentro de uma busca local muito mais veloz em comparação com k buscas locais rodando dentro de um reinício aleatório;

3. Obtendo alto desempenho

→ Vamos ilustrar os tipos de problemas que podem ser abordados para obter melhor perfomânce, quando se modifica o ILS;
→ Existem quatro componentes a serem considerados:

1. GenerateInitialSolution;
2. LocalSearch;

3. Perturbation;
4. AcceptanceCriterion;

$\rightarrow$ Antes de buscar desenvolver um algoritmo que alcance um ”nível artístico”, é interessante desenvolver uma versão mais básica do ILs;

$\rightarrow$ De fato, considera-se:
  (i) Pode-se iniciar com solução aleatória ou retornada por uma heurística de construção gulosa;
  (ii) Para a maioria dos problemas, uma busca local já está disponível;
  (iii) Para perturbação, um movimento aleatório dentro da vizinhança, de ordem maior do que a usada pelo algoritmo de busca local, pode ser surpreendentemente efetivo;
  (iv) Um primeiro palpite razoável para o critério de aceitação é forçar o custo dimiuir, correspondendo à uma primeira melhoria de descendência em $S*$;

$\rightarrow$ Implementações básicas, assim, do ILS, possuem performânce muito melhor do que abordagens de reinício aleatório;

$\rightarrow$ O desenvolvedor pode rodar esse ILS básico, afim de melhor sua intuição e tentar melhorar a performânce geral do algoritmo por melhorar seus módulos;

$\rightarrow$ Isso pode ser particularmente efetivo quando considerado as nuances específicas do problema considerado;

$\rightarrow$ Na prática é mais fácil fazer isto para o ILS do que para outros algoritmos;

$\rightarrow$ A razão, pode ser em reduzir-se a complexidade do ILS por sua modularidade;

$\rightarrow$ Além da função de cada componente ser de fácil entendimento;

$\rightarrow$ Por fim, a ultima tarefa a ser considerada é a otimização geral do algoritmo ILS;

$\rightarrow$ De fato, os diferentes componentes afetam uns aos outros, por isso é importante a compreensão de suas iterações;

$\rightarrow$ No entanto, essas iterações depende de cada problema abordado;

$\rightarrow$ O desenvolvedor tem o poder de escolha da sua implementação e melhoria;

$\rightarrow$ Sem nenhuma melhoria, o ILS é simples, fácil de implementar e uma meta-heurística eficiente;

$\rightarrow$ Mas, se trabalhos, os quatro componentes do ILS, tornam o algoritmo competitivo e possibilita chegar ao ”nível de arte”;

3.1. Solução inicial

$\rightarrow$ Aplicar a busca local à solução incial s_0, nos dá o ponto inicial S_0* do caminho no conjunto $S*$;

$\rightarrow$ Iniciar com um bom S_0* pode ser importante caso deseja-se obter soluções de alta qualidade o mais rápido possível;

$\rightarrow$ Escolhas padrão para s_0 podem ser uma solução incial aleatória ou uma solução retornada por uma heurística de construção gulosa;

$\rightarrow$ Uma solução gulosa possui duas vantagens em relação à solução inicial aleatória:
  (i) Quando combinadas com uma busca local, soluções gulosas, resultam em um s_0* de melhor qualidade;
  (ii) Uma busca local partindo de uma solução gulosa necessita de menos passos de melhoria e a busca local requer menos tempo de CPU;

$\rightarrow$ Para grandes tempos de computação, a dependência de s_0 na solução final do ILS é refletida quão rápidamente a memória da solução inicial é perdida, quando performado um caminho $S*$;

3.2. Perturbação

- $\rightarrow$ A principal desvantagem do local descent é a de ficar preso em um mínimo local que é significantemente pior do que o ótimo global;
- $\rightarrow$ Muito similar ao Simulated Annealing, o ILS escapa do ótimo local, aplicando o perturbações ao local mínimo atual;
- $\rightarrow$ Nomei-se *força* da perturbação o número de componentes de uma solução que são modificados;
- $\rightarrow$ Para o TSP, é o número de arestas quesão modificadas num caminho;
- $\rightarrow$ Para o FSP, é o número de tarefas que são movidas na perturbação;
- $\rightarrow$ Geralmente, a busca local não pode desfazer a perturbação, uma vez que retornaria para o ótimo local;
- $\rightarrow$ Bons resultados podem ser obtidos por pereturbações que levam em conta propriedades do problema e são bem combinadas com o algoritmo de busca local;
- $\rightarrow$ Quanto uma perturbação deveria mudar uma solução atual?
- $\rightarrow$ Se a perturbação for muito forte, o ILS deve comportar-se como reinicío aleatório;
- $\rightarrow$ Assim, melhores soluções só serão encontradas com baixa probalidade;
- $\rightarrow$ Se a perturbação for muito pequena, a busca local irá retornar, recorrentemente, para o ótimo local já visitado previamente;
- $\rightarrow$ Assim, a diversificação do espaço de busca será muito limitada;
- $\rightarrow$ Um exemplo de uma perturbação, simples porém efetiva, para o TSP é o *double-bridge move*;
- $\rightarrow$ Esta perturbação ”corta” quatro arestas (de força 4), e introduz quatro novas arestas;
- $\rightarrow$ Observe que cada ponte é de *2-change*, mas nenhum das *2-change* individualmente mantem o caminho conectado;
- $\rightarrow$ Aproximadamente, todos os estudos do ILS para o TSP, tem incorporado esse tipo de perturbação e tem sido efetivo para instâncias de todos os tamanhos;
- $\rightarrow$ Isso, porque muda a topologia do caminho e pode operar em quadruplas de cidades distantes;
- $\rightarrow$ Enquanto a busca local sempre modifica o caminho entre cidades próximas;
- $\rightarrow$ Pode-se utilizar uma busca local mais poderosa, que incluam mudanças do tipo *double-bridge*, no entanto, computacionalmente inviável, pelo alto custo em relação aos métodos de busca local, utilizado atualmenete;
- $\rightarrow$ Efetivamente, a perturbação *double-bridge*, não pode ser desfeita facilmente, nem mesmo por algoritmos simples busca local como 2-opt e 3-opt, ou pelo *Lin-Kernighan*, que é normalmente o campião entre algoritmos de busca local para o TSP;
- $\rightarrow$ Além disso, essa perturbação muito o tamanho do caminho;
- $\rightarrow$ Assim, se a solução atual é muito boa, é quase certeza que a próxima será boa também;
- $\rightarrow$ Essas duass propriedades da perturbação:
  - (i) A sua força pequena;
  - (ii) A diferença fundamental na sua natureza, entre as mudanças utilizadas na busca local;
- $\rightarrow$ Fazem com que o TSP seja a aplicação perfeita para o ILS;
- $\rightarrow$ Mas, para outros problemas, achar uma perturbação efetiva é mais difícil;

→ Para problemas como o TSP é de se esperar um ILS satisfatório, quando utilizado perturbações de tamanho fixo (independente do tamanho da instância);
→ Ao contrário de problemas mais difíceis, perturbações diretas podem levar à uma performance pobre;
→ Claramente, a força da perturbação não é todo o problema;
→ A sua natureza é tão importante quanto e merece ser discutida;
→ Finalmente, é importante pensar na velocidade;
→ Perturbações fracas, levam à execuções mais rápidas do *LocalSearch*;
→ Todos estes aspéctos devem ser levados em conta quando pretende-se otmizar esse módulo;

3.2.1. Força da perturbação

→ Para alguns problemas, uma força de perturbação apropriada é muito pequena e parece ser bem independente do tamanho da instância;
→ Esse é uma caso tanto para o TSP quanto para o FSP;
→ O ILS é bem competitivo, em relação à outras excelentes metahuerísticas;
→ Também pode-se considerar outros problemas, onde no lugar se é dirigido à perturbações de grande comprimento;

3.2.2. Perturbações adaptativas

→ O comportamento do ILS para problemas como o QAP e outros de otimização combinatória, mostram que que não existe, em princípio, um único tamanho melhor para aperturbação;
→ Isso motiva a possibilidade de modificar a força de perturbação e adatapta-la durante a sua execução;
→ Uma possibilidade de fazer isso, é explorar o histórico de buscas;
→ Para o desenvolvimento de tais esquemas, pode-se inspirar no contexto do *tabu serach*;
→ *Ver a propósta de Battitian and Prostasi*;
→ Outra maneira disto é mudar a força da perturbação determinísticamente durante a busca;
→ *Ver basic variable neighborhood search (basic VNS)*;
→ Em particular, ideias como *strategic oscillations* podem ser úteis para derivar perturbações efetivas;
→ Esse é também o espírito de algoritmos de busca reativas;

3.2.3. Esquemas de perturbações mais complexas

→ Perturbações podem ser mais complexas do que mudanças em vizinhanças de maior ordem;
→ Um procedimento geral para gerar s' do atual $s*$ segue-se:

1. Gentilmente, modifica-se a definição da instância, por exemplo, por meio dos parâmetros que definem a variação dos custos;

2. Para esta instância modificada, executar o *LocalSearch* usando $s*$ como entrada;
3. A saída é a solução perturbada s';

→ Interessantemente esse método foi proposto em trabalhos antigos do ILS;
→ Baxter e depois Codenotti et al., primeiro mudaram sutilmente as coordenadas das cidades;
→ Então, aplicaram a busca local para $s*$, usando essas localizações perturbadas das cidades;
→ Obtendo o novo caminho s';
→ Então, executando o *LocalSearch* em s' usando as coordenadas não perturbadas das cidades, encontram novos candidatos para o caminho $s*'$;
→ Outra maneira sofisticada de gerar boas perturbações consiste em otimizar uma sub-parte do problema;
→ Caso a tarefa seja muito complicada para a heurística embarcada, pode-se obter bons resulados;
→ Esse esquema funciona bem porque:
  (i) A busca local é incapaz de desfazer perturbações;
  (ii) Após aperturbação, as soluções tendem a ser muitos boas e ter ”novas” partes que são otimizadas;

3.2.4. Velocidade

→ No contexto de problemas ”simples” onde o ILS trabalha muito bem com perturbações fracas (de tamanho fixo);
→ Existe outra razão na qual esta metaheurística pode performar muito melhor do que o reinício aleatório: a velocidade;
→ De fato, a *LocalSearch* usualmente executará muito mais rápido em uma solução obtida pela aplicação de uma perturbação pequena em um ótimo local, do que em uma solução de início aleatório;
→ Como consequência, o ILS consegue executar muito mais buscas locais do que reinícios aleatórios em mesmo tempo de CPU;
→ Como exemplo qualitativo, consideremos o TSP Euclidiano:
  – $O(n)$ mudanças locais devem ser aplicadas pela busca local afim de obter um ótimo local de reinício aleatório;
  – Enquanto, aproximadamente, um número constante é necessário no ILS quando utilizado o s' obtido pela perturbação *double-bridge*;
→ Portanto, em um dado tempo de CPU, o ILS pode amostrar vários ótimos locais à mais do que o reinício aleatório pode;
→ Esse fator de velocidade, dá ao ILS uma vantagem considerável tendo em vista outros esquemas de reinício;

3.3. Critério de aceitação

→ O ILS realiza um caminho aleatório em $S*$, o espaço do mínimo local;
→ O mecanismo de perturbação conjuntamente com a busca local define a possibilidade de transição entre a solução atual $s*$ em $S*$, para uma solução vizinha $s*'$, também em $S*$;

- $\rightarrow$ O processo $AcceptanceCriterion$, então determina se $S*'$ é aceito ou não como nova soulção atual;
- $\rightarrow$ O $AcceptanceCriterion$ possui grande influência na natureza e efetividade da caminhada em $S*$.
- $\rightarrow$ Isso pode ser usado como base de controle para balancear entre intensificação e diversificação da busca;
- $\rightarrow$ Um modo simples de ilustrar isso é considerar um critério de aceitação Markoviano;
- $\rightarrow$ Uma intensificação forte é atingida somente se melhores soluções são aceitas;
- $\rightarrow$ Esse tipo de critério de aceitação é denominado como *Better* e definido para problemas de minização como:

$$Better(s*, s*', \textit{history}) = \begin{cases} s*' \text{ if } C(s*') < C(s*) \\ s* \text{ otherwise} \end{cases} \quad (1)$$

- $\rightarrow$ Em outro extremo existe o critério de aceitação "caminhada aleatória" (denotado como RW);
- $\rightarrow$ Esse critério sempre aplica a perturbação ao local ótimo visitado, independente do seu custo;

$$RW(s*, s*', \textit{history}) = s*' \quad (2)$$

- $\rightarrow$ Esse critério prioriza diversificação ao invés de intensificação;
- $\rightarrow$ Existem outras escolhas intermediárias entre esse extremos;
- $\rightarrow$ Um exemplo é o algoritmo *large-step Markov chain*, proposto por Martin, Otto e Felten;
- $\rightarrow$ Um tipo de critério de aceitação de *simulated annealing* é aplicado;
- $\rightarrow$ Chamado de *LSMC*, sempre aceita $s*'$ caso tenha custo melhor que $s*$;
- $\rightarrow$ Caso seja pior, aceita $s*'$ com uma probabiliade de $exp(C(s*) - C(s*'))/T$, onde T é a temperatura, que usualmente decresce durante a execução;
- $\rightarrow$ Um caso limitante para o uso de memória no critério de aceitação é o de completo reinício;
- $\rightarrow$ Utilizado quando a intnsificação para ter-se tornado ineficiente;
- $\rightarrow$ O critério de aceitação baseado nisso é chamado de *Restart*, definido como:

$$Better(s*, s*', \textit{history}) = \begin{cases} s*' \text{ if } C(s*') < C(s*) \\ s \text{ if } C(s*') <\geq C(s*) \text{ and } i - i_{last} > i_r \\ s* \text{ otherwise} \end{cases} \quad (3)$$

- $\rightarrow$ Onde i_r é um parametro que define se o algoritmo deve ser reiniciado, caso não haja melhoria dentro de i_r iterações;
- $\rightarrow$ Tipicamente, s pode ser gerado de diversas maneiras;
- $\rightarrow$ A eficiencia geral do ILS pode ser afetada de acordo com o critério de aceitação aplicado;

3.4. Busca Local

- $\rightarrow$ Diferentes heurísticas podem ser utilizadas;
- $\rightarrow$ Para soluções melhores, o iterated Lin-Kernighan é melhor do que o 3-opt, que por sua vez é melhor que o 2-opt;
- $\rightarrow$ No entanto, são inversamente mais rápidas em relação a sua qualidade de soluções;

3.5. Otimização global para o ILS

$\rightarrow$ Para uma otimização global do ILS, a otimização em um dos componetes depende dos demais;

$\rightarrow$ Como o componente $GenerateIntialSolution$ é executado somente uma vez e sua solução é perdida logo em seguida, não parece ser tão interessante a sua otimização;

$\rightarrow$ A melhor escolha para o $Perturbation$ depende da escolha do $LocalSearch$;

$\rightarrow$ Enquanto a escolha do $AcceptanceCriterion$ depende do $Perturbation$ e do $LocalSearch$;

$\rightarrow$ Na prática, a otimização global pode ser aproximada por otimizar sucessivamente cada compente, assumindo que os demais estão fixados;

$\rightarrow$ Isso não garante uma otimização global de fato, mas garante uma otimização geral do algoritmo;

3.5.1. Características da busca espacial

$\rightarrow$ Principais dependências dos componentes:

1. A perturbação não deve ser facilmente desfeita pela busca local;
2. Se a busca local tiver deficiências óbvias, uma boa perturbação pode compensar isso;
3. A combinação $Perturbation - AcceptanceCriterion$ determina o equilíbrio relativo à internsificação e diversificação;
4. Perturbações grandes são úteis somente se elas podem ser aceitas, que ocorre somente se o critério de aceitação não é tão enviezada à soluções melhores;

$\rightarrow$ Em linhas gerais, o $LocalSearch$ deve ser o mais poderoso possível contanto que não seja tão custoso em tempo de CPU;

$\rightarrow$ Escolha uma boa perturbação seguindo o que foi apresentado previamente;

$\rightarrow$ Finalmente defina a rotina $AcceptanceCriterion$ de maneira que $S*$ seja amostrado adequadamente;

$\rightarrow$ Desse ponto de vista a otimização geral do ILS é feita de baixo para cima, mas com iteração;

$\rightarrow$ Provavelmente o problema central é o que colocar na $Perturbation$;

$\rightarrow$ Pode-se restringir a perturbação para ser fraca?

$\rightarrow$ Do ponto de vista teórico isso depende de como as melhores soluções estão agrupadas em $S*$;

$\rightarrow$ Em alguns problemas, como o TSP, existe uma forte corelação entre o custo de uma solução e a sua ”distância” do ótimo;

$\rightarrow$ De fato, as melhores soluções estão agrupadas, i.e., possuem muitos componetes similares;

$\rightarrow$ Isto é referenciado de diversas maneiras: Fenômeno Maciço Central, princípio de proximidade da otimalidade e simetria de réplica;

$\rightarrow$ Se o problema considerado possui essa propriedade, não é irrasoável esperar encontrar o ótimo verdadeiro usando uma amostra com viés de $S*$;

$\rightarrow$ Em particular, é claro ser útil o uso de intensificação para atingir o ótimo global;

4. ILS para o TSP

→ O TSP é provavelmente o problema de otimização combinatória mais bem conhecido;
→ De fato, é um teste de mesa inicial para o desenvolvimento de novos algoritmos;
→ Uma boa performânce com o TSP é tido como evidência do valor de boas ideias;
→ COmo para muitas metahuerísticas, alguns dos primeiros algoritmos de ILS foram introduzidos e testados com o TSP;
→ Uma maior melhoria na performânce dos algoritmos de ILS vieram do algoritmo *large-step Markov chain* (LSMC), proprosto por Martin, Otto e Felten;
→ Utilizou-se um simulated annealing como critério de aceitação onde consideram tanto o 3-opt como a heurística Lin-Kernighan (LK) que são os melhores algoritmos de busca local para o TSP;
→ O ingrediente chave para para o trabalho deles, foi o uso do double bridge move para perturbação;

References