Aluno: Rodrigo Marins Esteves

Problema do Caixeiro Viajante Assimétrico

Descrição: Um caixeiro pretende passar por n cidades percorrendo o menor caminho possível. Para isso, presumem-se as seguintes afirmativas:

1. Serão selecionadas uma cidade inicial e uma cidade final, onde a viagem respectivamente se iniciará e se encerrará.
2. O caixeiro irá passar apenas uma vez por cada cidade.
3. Todos as cidades possuem conexões entre si, onde tais conexões possuem suas respectivas distancias.
4. As distancias entre duas cidades A e B podem ser diferente se considerarmos de qual delas iremos partir e em qual iremos chegar, ou seja, a distância de A para B pode ser diferente da distancia de B para A.

Modelagem: O problema será modelado em grafos, onde cada vértice representará uma cidade e cada aresta será um caminho entre tais cidades. Cada aresta, portanto, terá seu peso, que por sua vez representará a distancia entre as cidades.

Bibliografia: http://www.mat.ufrgs.br/~portosil/caixeiro.html

Repositórios:

- https://wwwproxy.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/atsp/
- https://projects.coin-or.org/metslib/browser/Examples/trunk/atsp/data/ry48p.atsp

Heurística Construtiva:

- Heurística do Vizinho mais Próximo (Nearest Neighbor Heuristic)

  Seja G(V,E) um grafo completo, onde V são os vértices e E as arestas.

  1. Selecione um v ∈ V como nó inicial.
  2. Encontre o menor e ∈ E que leve a um u ∈ V que não tenha sido visitado.
  3. Selecione u como o vértice atual.
  4. Marque u como visitado.
  5. Se todos os nós de V foram visitados, termine.
  6. Volte ao passo 2.

Primeira implementação

O primeiro trabalho consistiu na implementação de duas heurísticas construtivas para resolver o ATSP. Para este trabalho, foram implementadas a Heurística do Vizinho mais Próximo e a Heurística do Vizinho mais Próximo com transição dupla (construindo o caminho através dos dois nós mais extremos que já estão na solução). Foi também feita a randomização do primeiro algoritmo citado. Apesar do segundo ter sido mais eficiente, como será mostrado, achei mais prudente utilizar o primeiro para a randomização pois ele cabe melhor nas comparações que mostrarei adiante. Os algoritmos foram construídos em C e para a randomização foi utilizada uma implementação do algoritmo Mersenne Twister encontrada na internet.

- Heurística do Vizinho mais Próximo x Heurística do Vizinho mais Próximo Bidirecional

Como podemos ver na imagem abaixo, a heurística bidirecional apresentou uma pequena vantagem em relação à solução nos 3 arquivos testados.

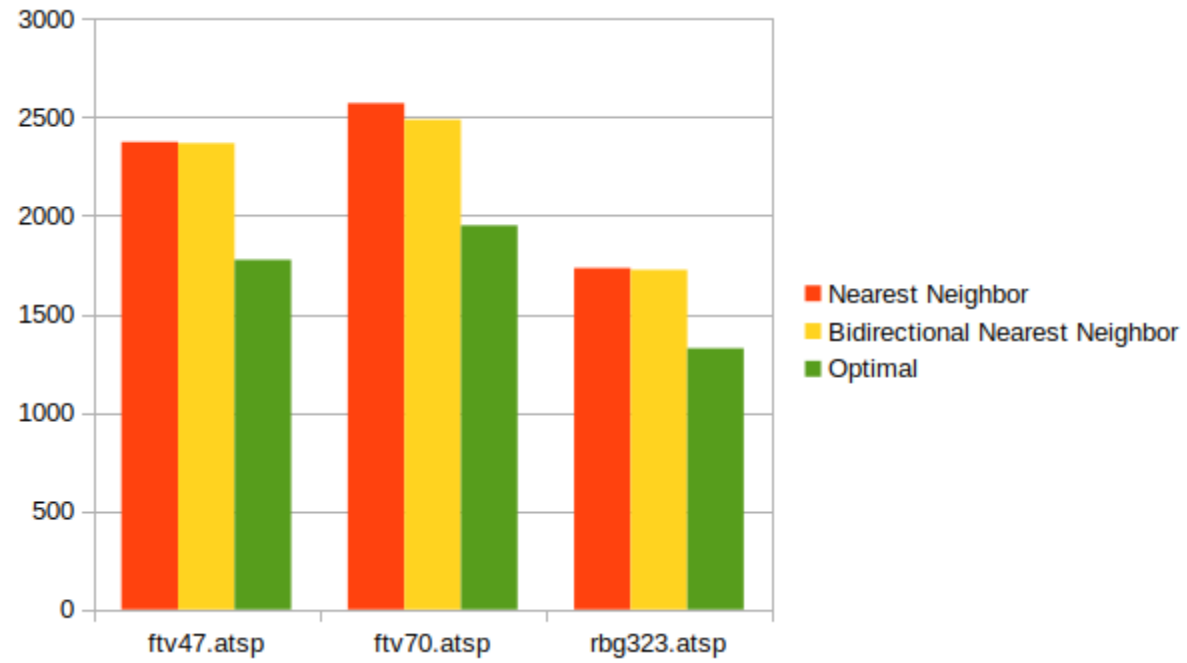

Fig1: (Heuristica/Arquivo x Tamanho da solução).

- Randomização da Heurística da Vizinho mais Próximo

Para testar o Algoritmo Randomizado, foram feitas 100 execuções em um grafo contendo 323 nós, todos conectados uns aos outros. Como podemos ver na imagem abaixo, a randomização pode trazer boas ou péssimas soluções. Ao aproximar o alpha de 0, temos soluções tendendo às soluções de um Algoritmo Guloso. Ao aproximar o alpha de 1, temos soluções que tendem às soluções de um algoritmo completamente randômico. Em alguns casos, com alpha = 0.1, foram encontradas algumas soluções melhores que a solução do Algoritmo Guloso, como pode ser visto na figura 3.

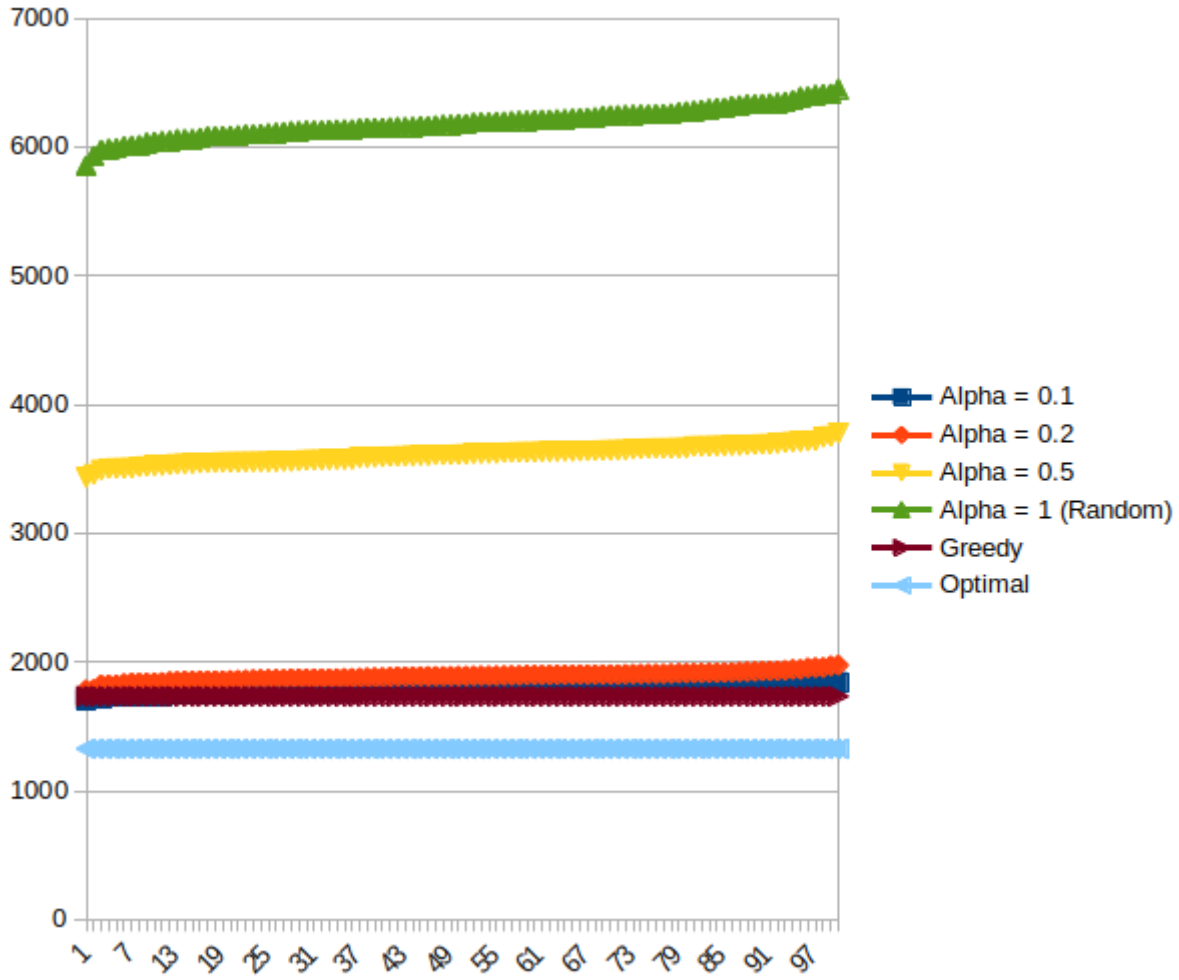

Figura 2: (Execuções x Valor das soluções).

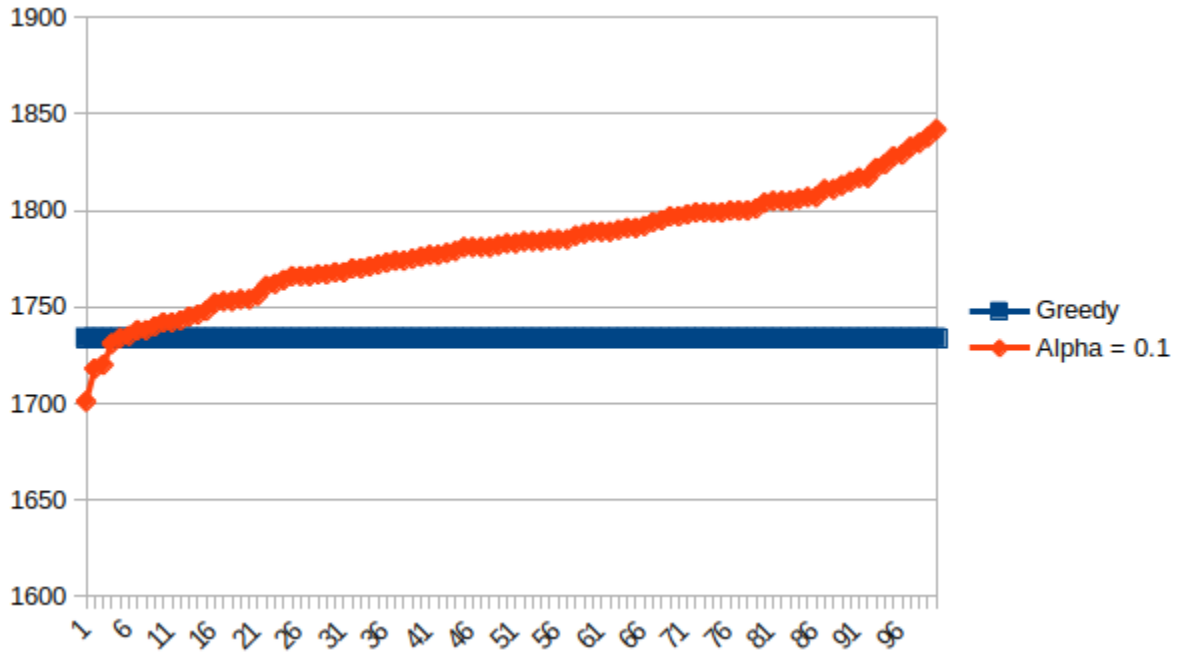

Figura 3: (Execuções x Valor das soluções). Comparação entre o Algoritmo Guloso e o Randomizado com aplha = 0.1. Pode se notar que em alguns casos o Algoritmo Randomizado encontra soluções melhores.

Segunda implementação

O segundo trabalho consistiu na implementação de algoritmos de Busca Local para, a partir de uma solução inicial calculada pelo Algoritmo Guloso, encontrar soluções melhores na vizinhança da solução inicial. Foram implementados dois algoritmos de Busca Local diferentes, que divergem justamente no conceito de vizinhança. O primeiro algoritmo utiliza o algoritmo 2-opt para selecionar uma vizinhança. O segundo algoritmo utiliza um conceito de vizinhança baseado na troca de posições de nós que distam entre zero e quatro nós um dos outros (analisando do nó atual para frente). Para exemplificar o funcionamento dessa segunda vizinhança, considere o seguinte:

> Seja **A-B-C-D-E-F-G-H-I-J-A** uma solução inicial.
> Se iniciarmos nossa análise em **B**, teríamos as possíveis soluções vizinhas:
>
> **A-C-B-D-E-F-G-H-I-J-A**
> **A-D-C-B-E-F-G-H-I-J-A**
> **A-E-C-D-B-F-G-H-I-J-A**
> **A-F-C-D-E-B-G-H-I-J-A**
> **A-G-C-D-E-F-B-H-I-J-A**
>
> Essa mesma análise, então, prossegue por todos os nós (quando possível).

Para tais algoritmos de Busca Local, foram implementadas tanto a estratégia Mais Aprimorante quando a estratégia Primeira Aprimorante.

Abaixo podemos ver o pseudocódigo do algoritmo 2-opt com estratégias melhor aprimorante e primeira aprimorante respectivamente. Para o algoritmo de busca local alternativo proposto, o pseudocódigo não difere muito, exceto pela escolha da vizinhança que se dá pelo critério mostrado acima.

```
best_solution = greedy()
Repeat until there is no improving to be made {
      for (i = 1; i < number of nodes- 1; i++) {
         for (k = i + 1; k < number of nodes; k++) {
            new_route = 2optSwap(existing_route, i, k)
            new_distance = calculateTotalDistance(new_route)
            if (new_distance < best_distance) {
               current_solution = new_route
               best_distance = new_distance
            }
         }
      }
      best_solution = current_solution
}
```

Algoritmo 2-opt com estratégia melhor aprimorante

```
best_solution = greedy()
Repeat until there is no improving to be made {
    start_again
    for (i = 1; i < number of nodes- 1; i++) {
        for (k = i + 1; k < number of nodes; k++) {
            new_route = 2optSwap(existing_route, i, k)
            new_distance = calculateTotalDistance(new_route)
            if (new_distance < best_distance) {
                existing_route = new_route
                best_distance = new_distance
                goto start_again
            }
        }
    }
}
```

Algoritmo 2-opt com estratégia primeira aprimorante

Análise:

Os algoritmos foram executados em 10 instancias do ATSP. A princípio, foi feita uma análise sobre como o tempo de processamento cresce nesses algoritmos de acordo com o crescimento da dimensão do problema. Como podemos ver na figura 1, para dimensões maiores, o tempo de execução do algoritmo 2-opt cresce desproporcionalmente em relação aos outros. Por tais motivos, as instancias testadas com dimensões grandes serão eliminadas das próximas análises de tempo com o fim de obter gráficos que sejam visualmente analisáveis. Além disso, as instancias com resultados (caminhos) de valores grandes demais também serão eliminadas das análises pelo mesmo motivo.

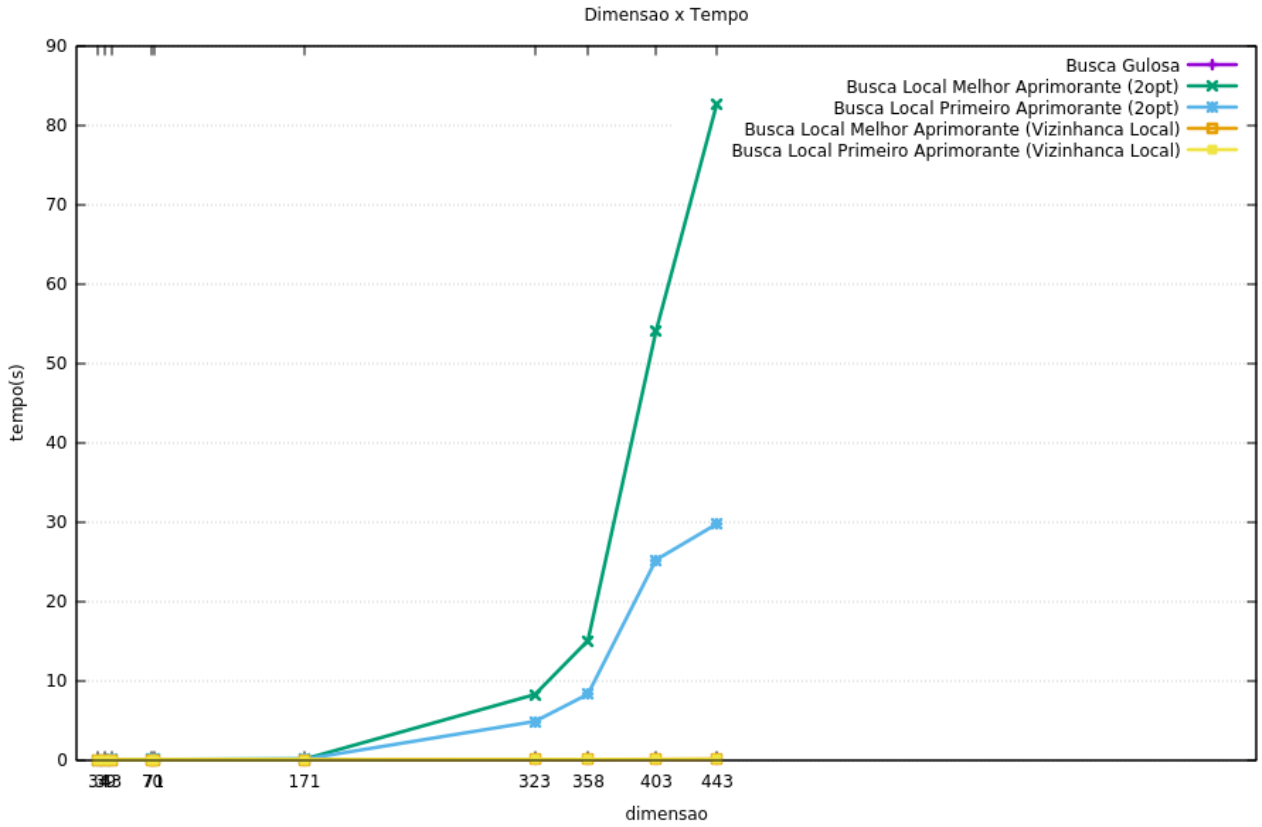

Figura 1: O crescimento do tempo de processamento dos algoritmos de acordo com o crescimento da dimensão do problema.

Ao excluirmos as duas instancias com maior dimensão, ainda assim temos um gráfico extremamente desproporcional, como pode ser visto na figura 2.
Devido a isso, foram removidas mais duas instâncias desta análise, sobrando então, 6 instancias. Como pode ser visto na figura 3, os algoritmos 2-opt sempre levam um tempo visivelmente maior para serem executados. A estratégia melhor aprimorante, em grande parte dos casos, também se apresentou como computacionalmente mais complexa. O outro algoritmo de busca local, baseado em uma vizinhança construída por troca de nós, se mostrou mais complexo em relação à busca gulosa, porém muito menos complexo que o algoritmo 2-opt.

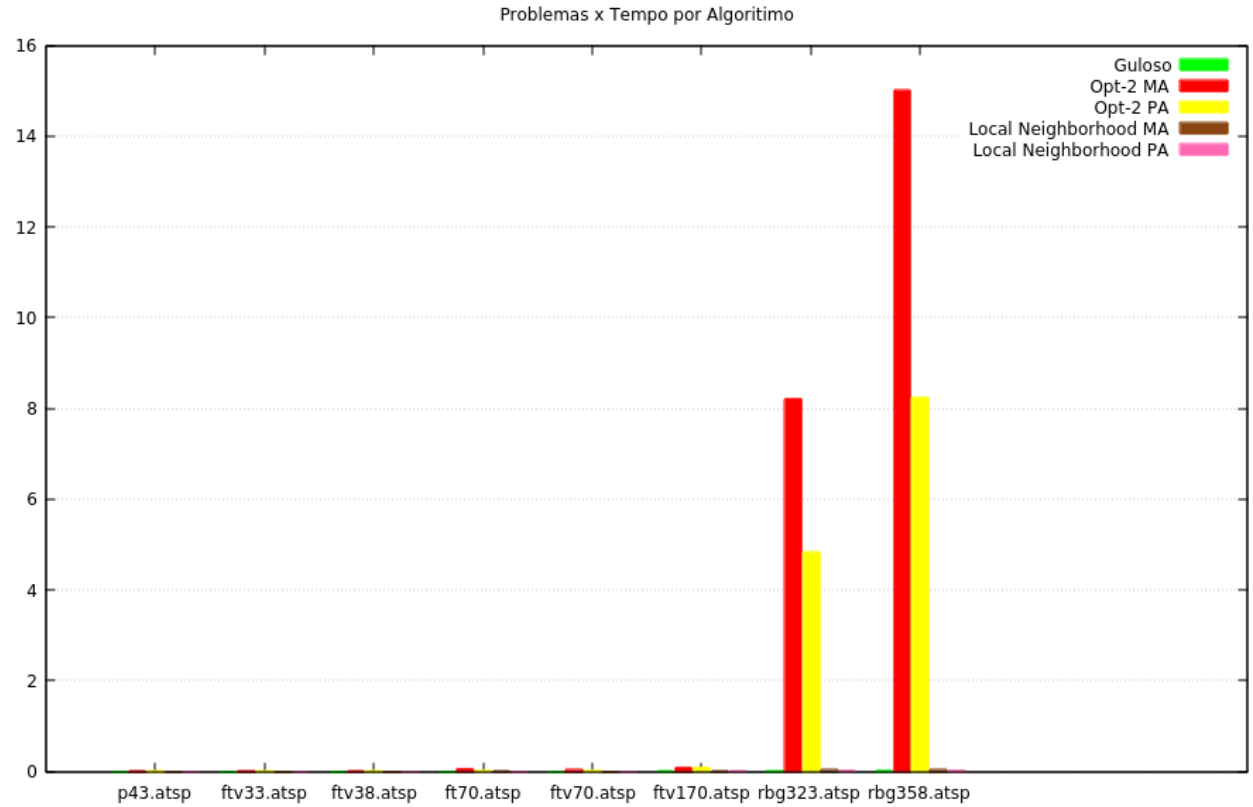

Figura 2: Instâncias grandes tornam a análise mais difícil quando analisadas junto a instâncias pequenas

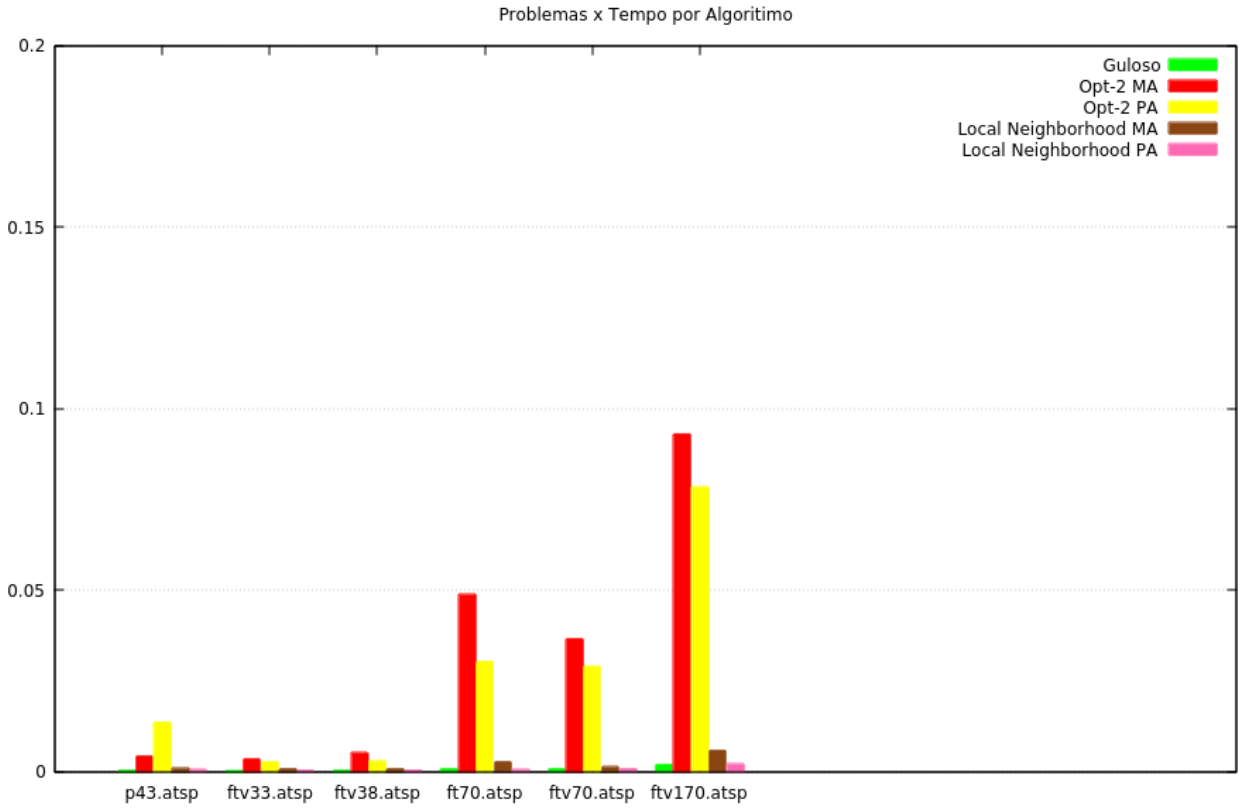

Figura 3: O algoritmo 2-opt se mostrou visivelmente mais complexo.

Dadas as considerações feitas sobre o tempo de execução, podemos agora analisar os algoritmos baseando-se nos resultados obtidos. Como pode ser visto na figura 4, o algoritmo 2-opt se mostrou mais eficiente que o outro algoritmo de busca local, e também mais eficiente do que o algoritmo guloso (o que deveria ocorrer por definição). Foi também constatado que para o algoritmo 2-opt, houve uma melhor performance utilizando-se a estratégia “primeira aprimorante” na maior parte dos casos. Em alguns casos específicos, nota-se também que o algoritmo de busca local cuja vizinhança foi construída pela troca da posição dos nós na solução, obteve resultados muito parecidos com os resultados do algoritmo 2-opt.

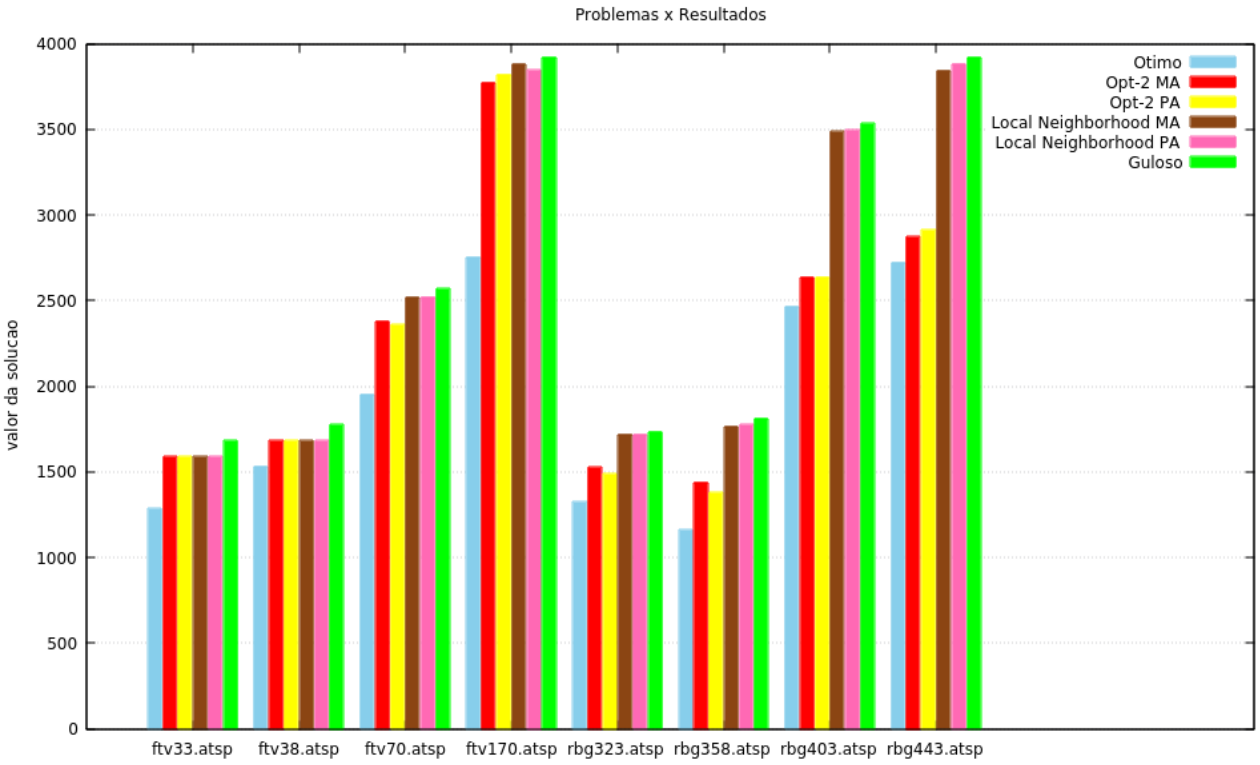

Figura 4: Valor da solução para cada problema por algoritmo. Foram excluídas das análise duas soluções cujas soluções grandes distorciam o gráfico.

Abaixo, podemos ver 4 gráficos utilizando as 10 instancias testadas que mostram isoladamente a solução ótima do problema, o resultado do algoritmo guloso, e o resultado do algoritmo de busca local e estratégia de seleção em questão. A figura 5 mostra, para todos as instâncias, o valor da solução ótima, o valor da solução da busca gulosa e o valor da solução obtida pelo algoritmo 2-opt com estratégia melhor aprimorante. A figura 6 mostra, para todos as instâncias, o valor da solução ótima, o valor da solução da busca gulosa e o valor da solução obtida pelo algoritmo 2-opt com estratégia primeira aprimorante. A figura 7 mostra, para todos as instâncias, o valor da solução ótima, o valor da solução da busca gulosa e o valor da solução obtida pelo algoritmo de busca local alternativo com estratégia melhor aprimorante. A figura 8 mostra, para todos as instâncias, o valor da solução ótima, o valor da solução da busca gulosa e o valor da solução obtida pelo algoritmo de busca local alternativo com estratégia primeira aprimorante. Como explicitado anteriormente, ao

juntar todas as instâncias os resultados não podem ser vistos em detalhes devido à escala, que se adapta ao maior valor, que por sua vez diverge muito dos outros.

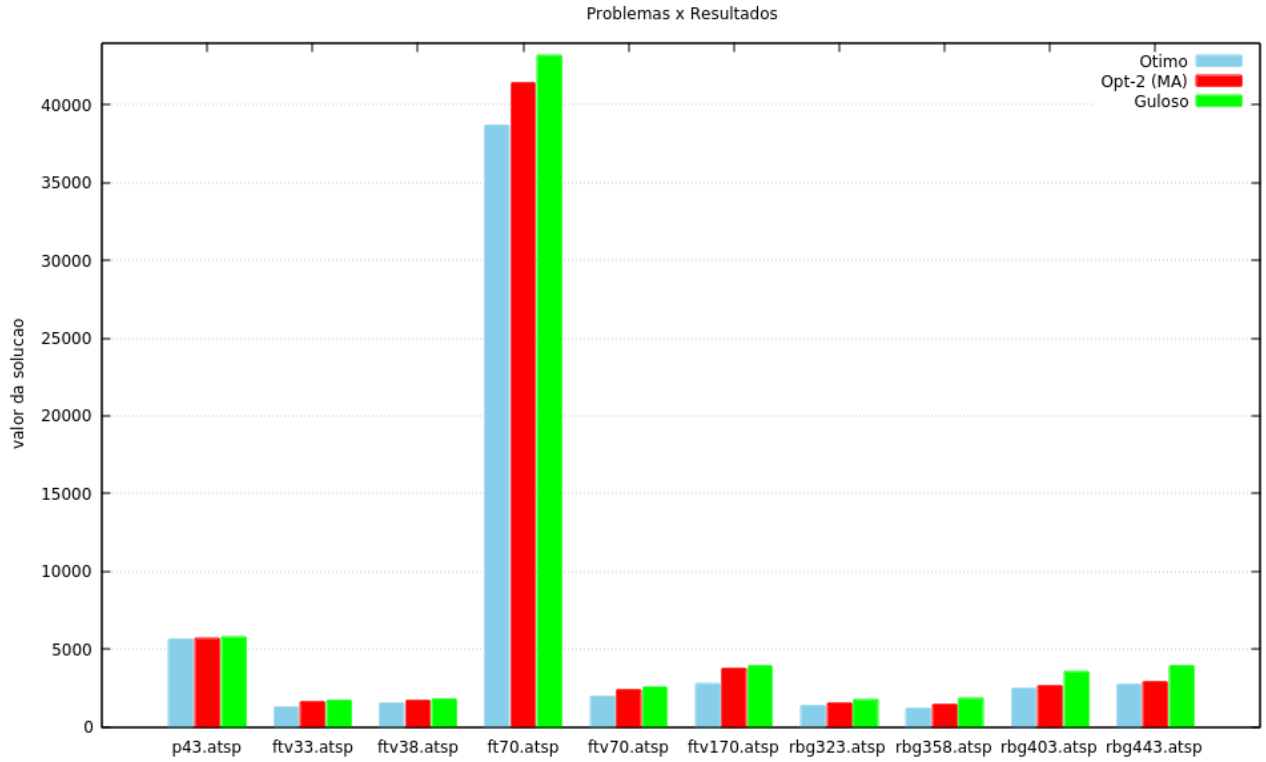

Figura 5

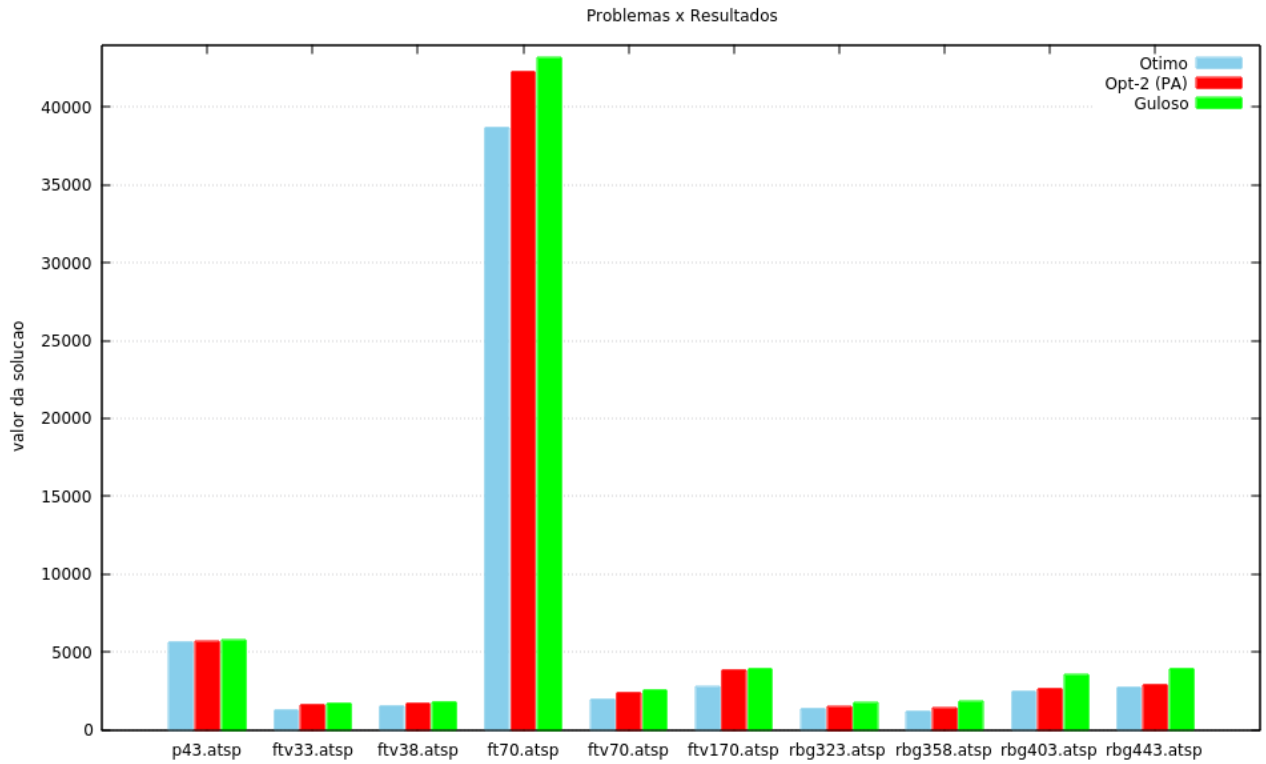

Figura 6

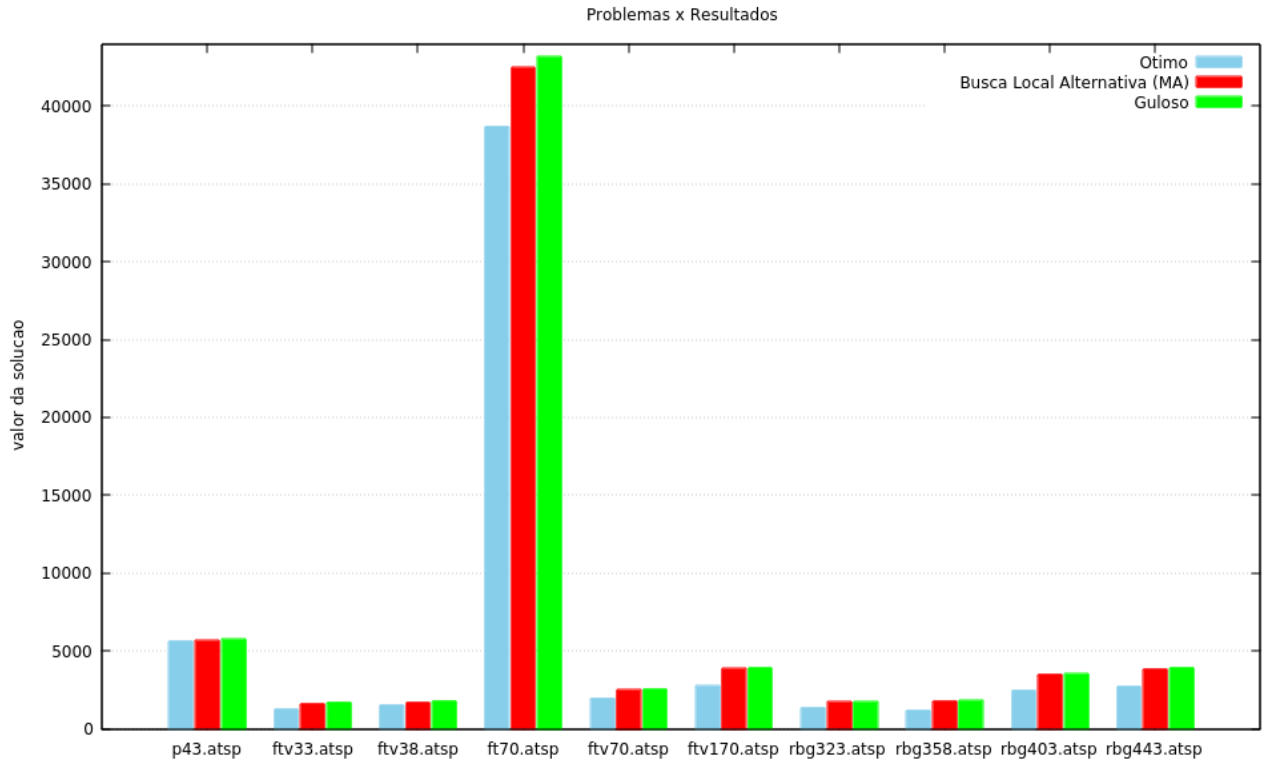

Figura 7

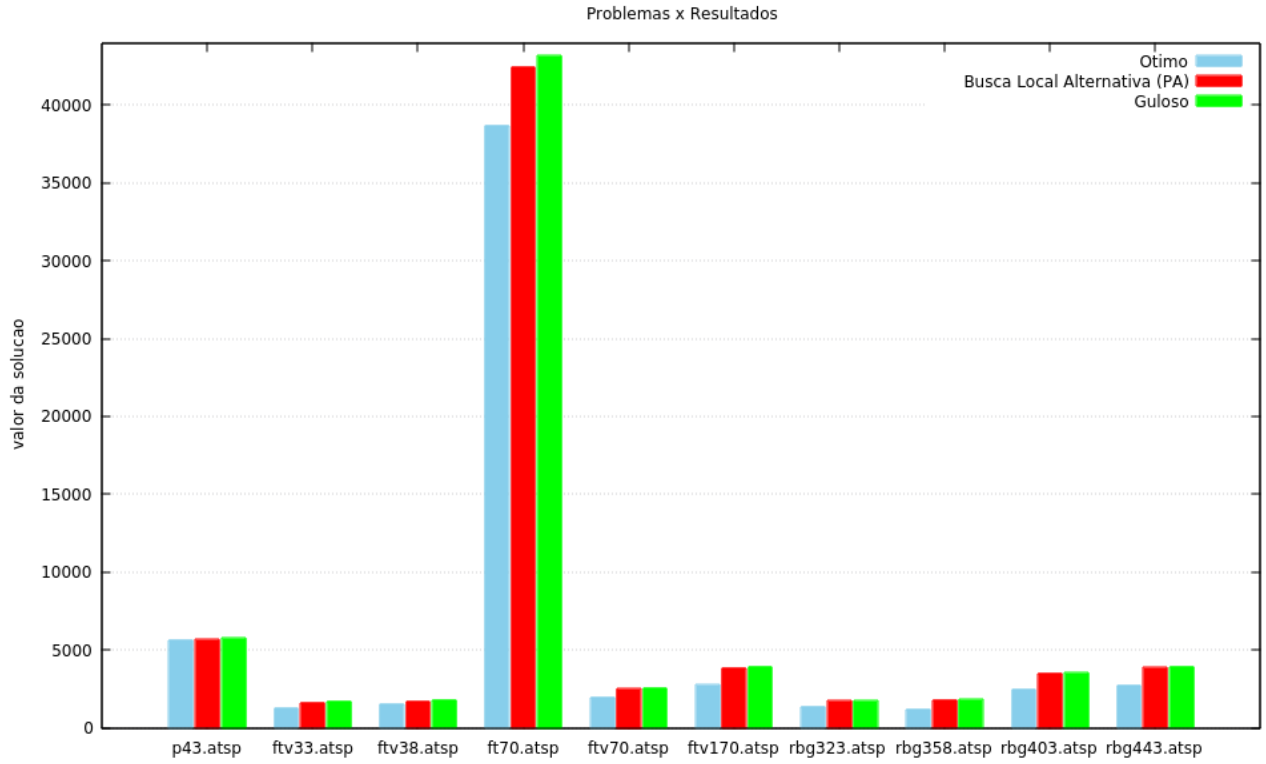

Figura 8

Terceira implementação

O terceiro trabalho consistiu na implementação do método GRASP através da Busca Local e do Método Semi-Guloso Randomizado já implementados anteriormente. A Busca Local utilizada utiliza uma vizinha gerada pelo algoritmo 2-opt com estratégia de busca mais aprimorante, visto que essa vizinhança se apresentou como mais efetiva nas análises da segunda implementação.
A princípio, o objetivo foi realizar testes com algumas instâncias do problema para identificar qual seria o melhor valor de alpha para a randomização

Para 5 instâncias, foi feito o seguinte processo:

1. *Execute 200 iterações do GRASP com aplha = 0.1*
2. *Meça o tempo de execução*
3. *Para cada x pertencente ao conjunto [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9], faça alpha = x e execute o GRASP até o tempo de execução atingir o tempo medido.*

Ao fim do processo, os dados da execução foram salvos em arquivos, que posteriormente foram analisados.. Como pode ser visto nas figuras abaixo (Figura1 à Figura10), na maior parte dos casos, os melhores “melhor resultado” e “resultado médio” são obtidos quando alpha é igual a 0.1. Também fica claro através dos gráficos que alpha = 0.1 leva o algoritmo a boas soluções em um tempo muito menor. Em um caso específico, utlizar alpha = 0.2 trouxe resultados melhores. Em alguns casos, pode-se notar que alphas mais próximos de 1 em certos momentos acham resultados bons, talvez pelo fato de poder enumerar um número maior de possíveis soluções que serão escolhidas ao acaso, o que gera soluções mais distintas entre si, o que consequentemente permite que a busca local explore mais vizinhanças.

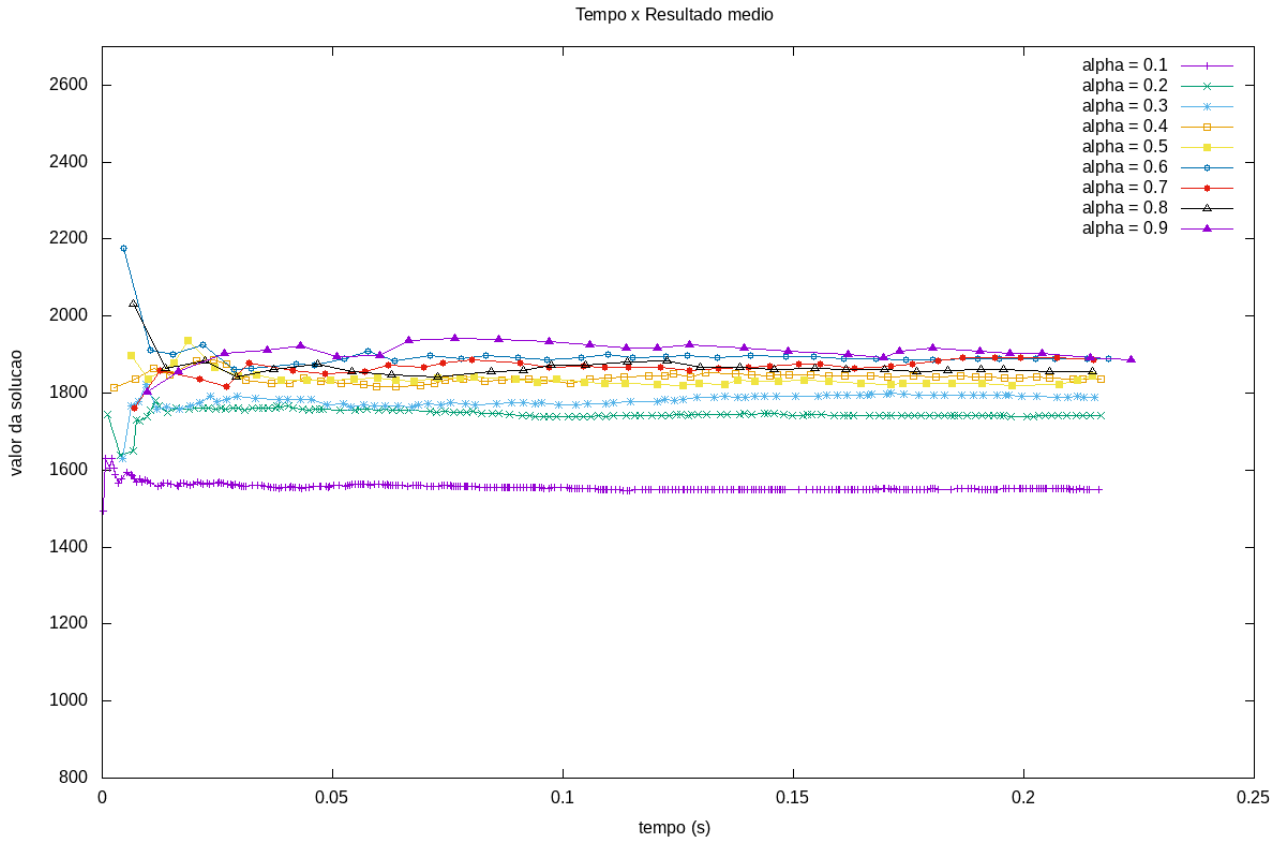

Figura 1: Solução média para a instância ftv33

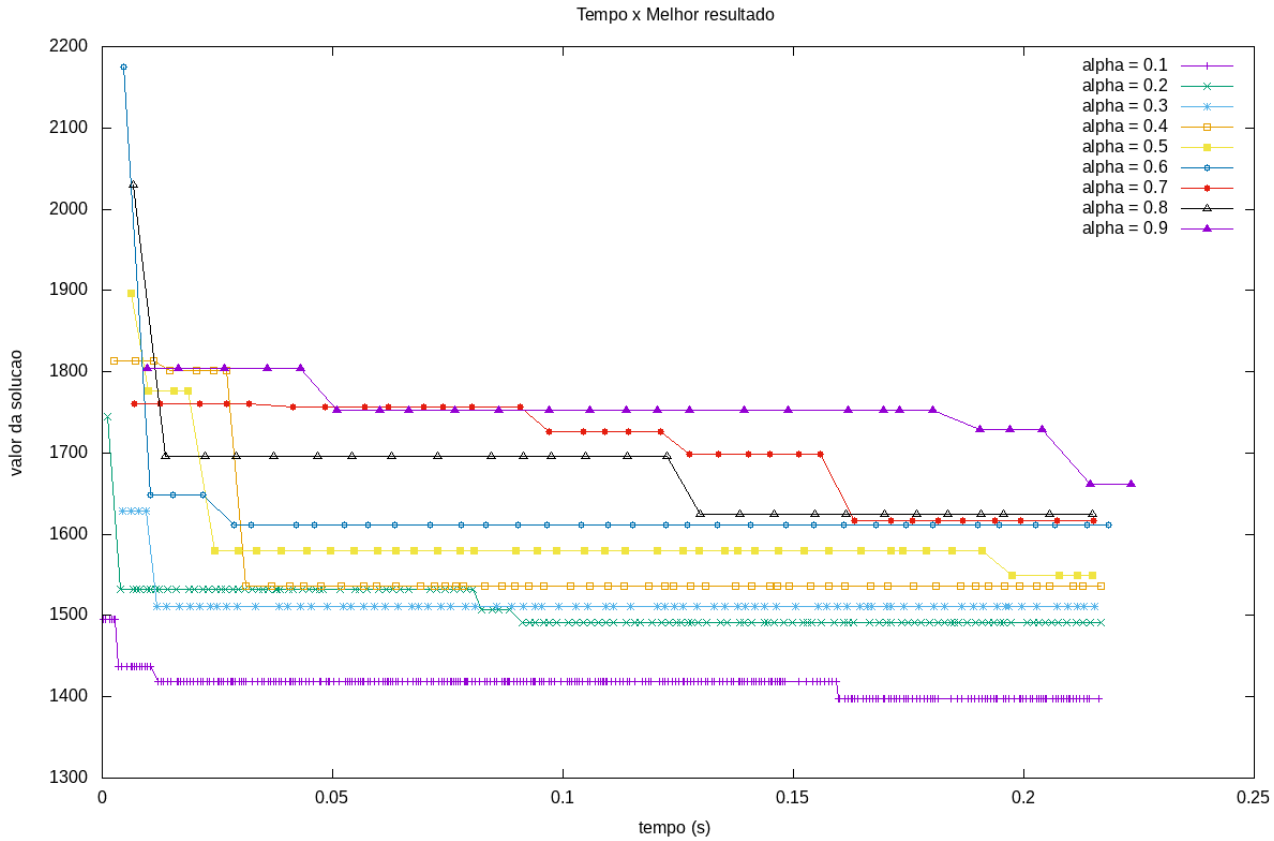

Figura 2: Melhor solução para a instância ftv33

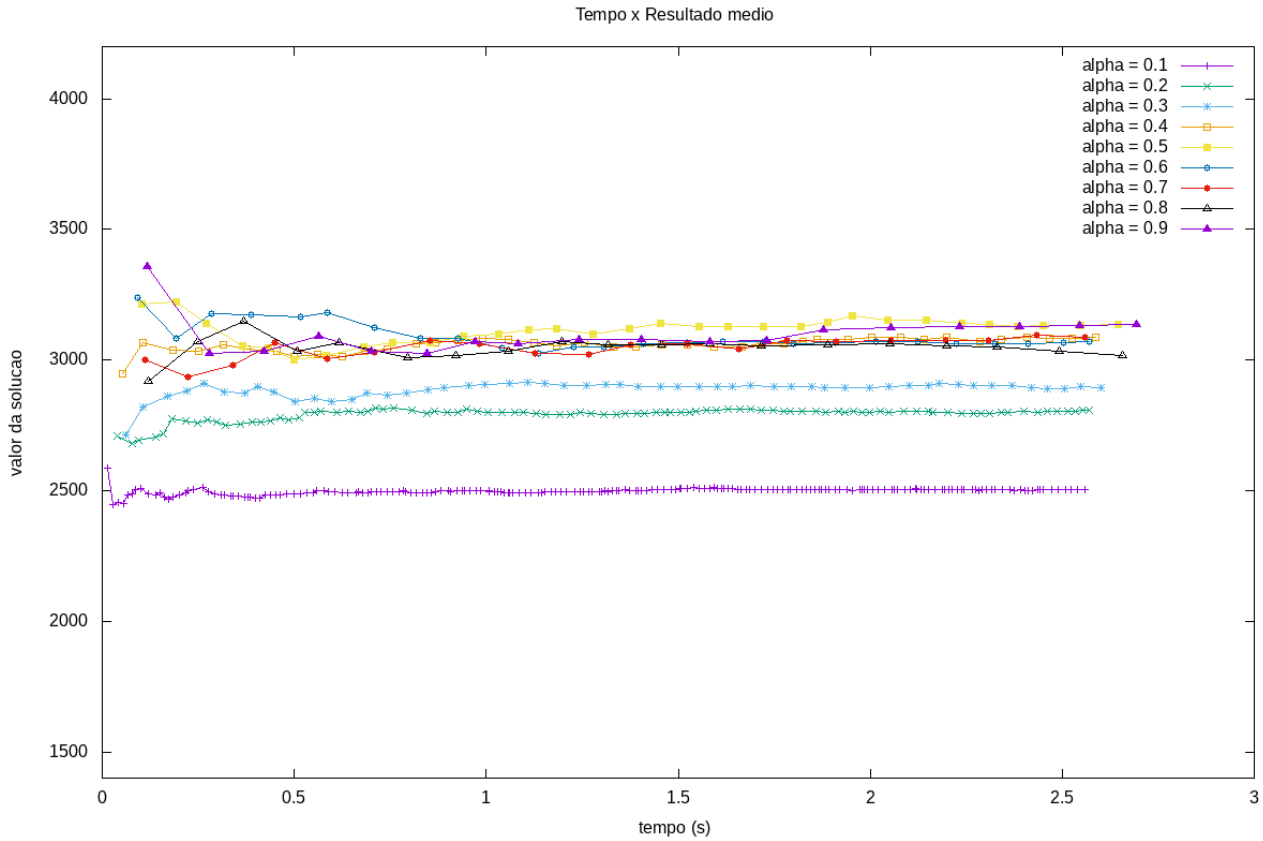

Figura 3: Solução média para a instância ftv64

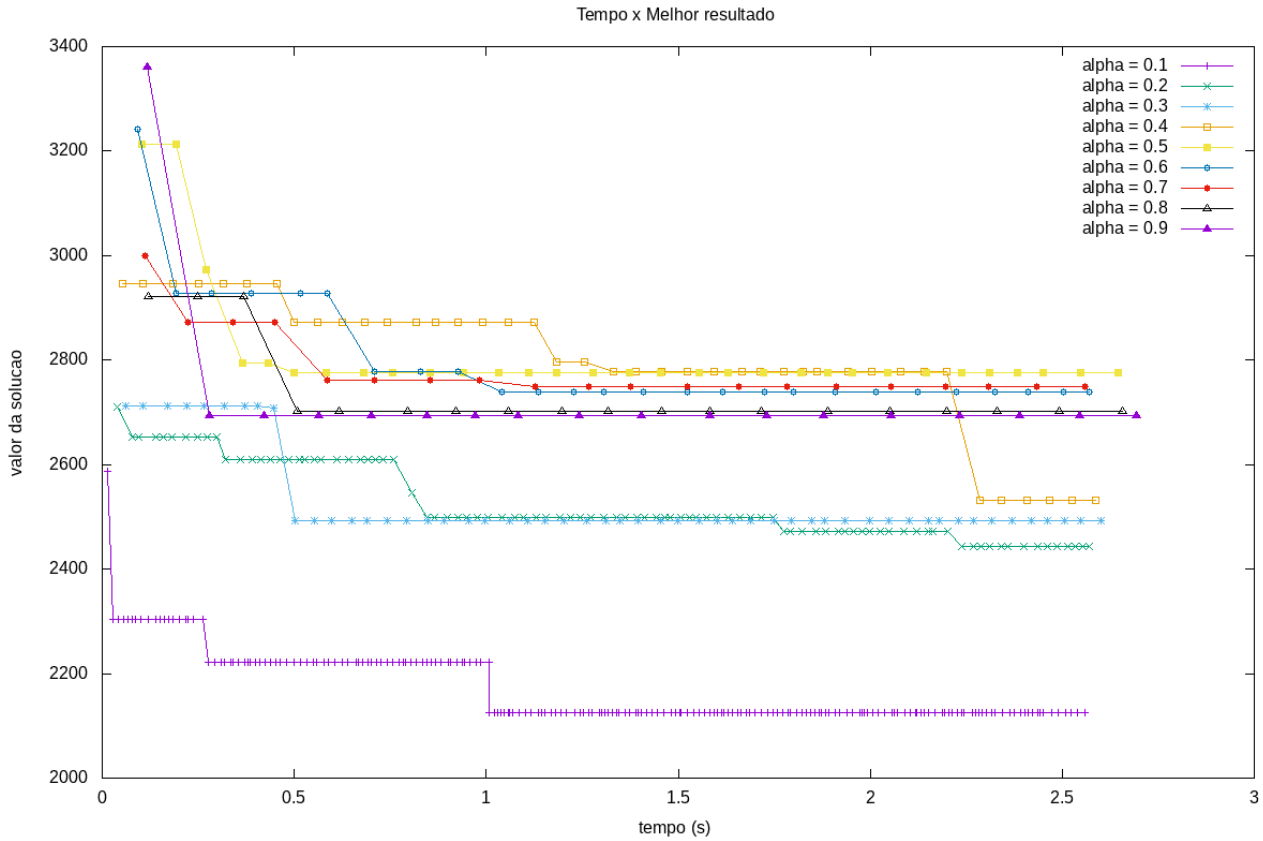

Figura 4: Melhor solução para a instância ftv64

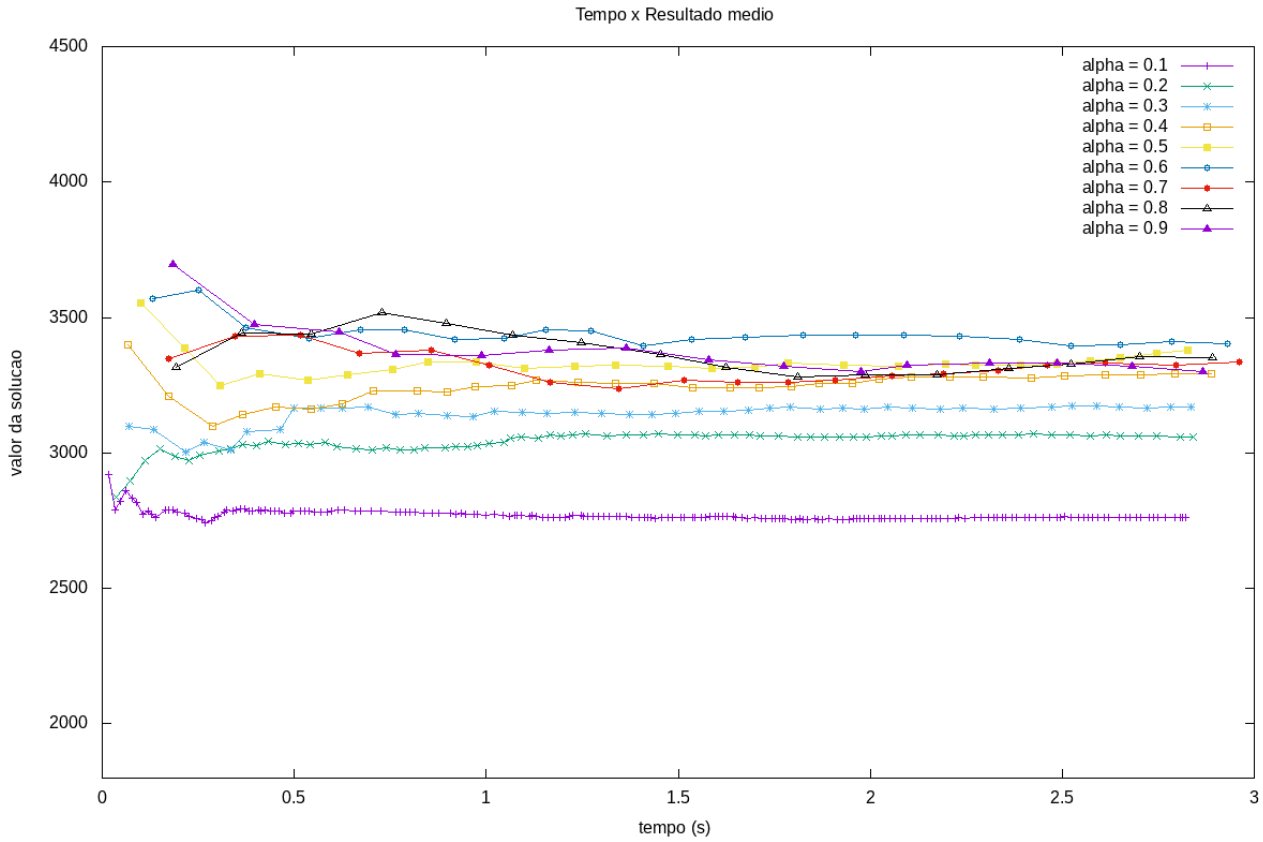

Figura 5: Solução média para a instância ftv70

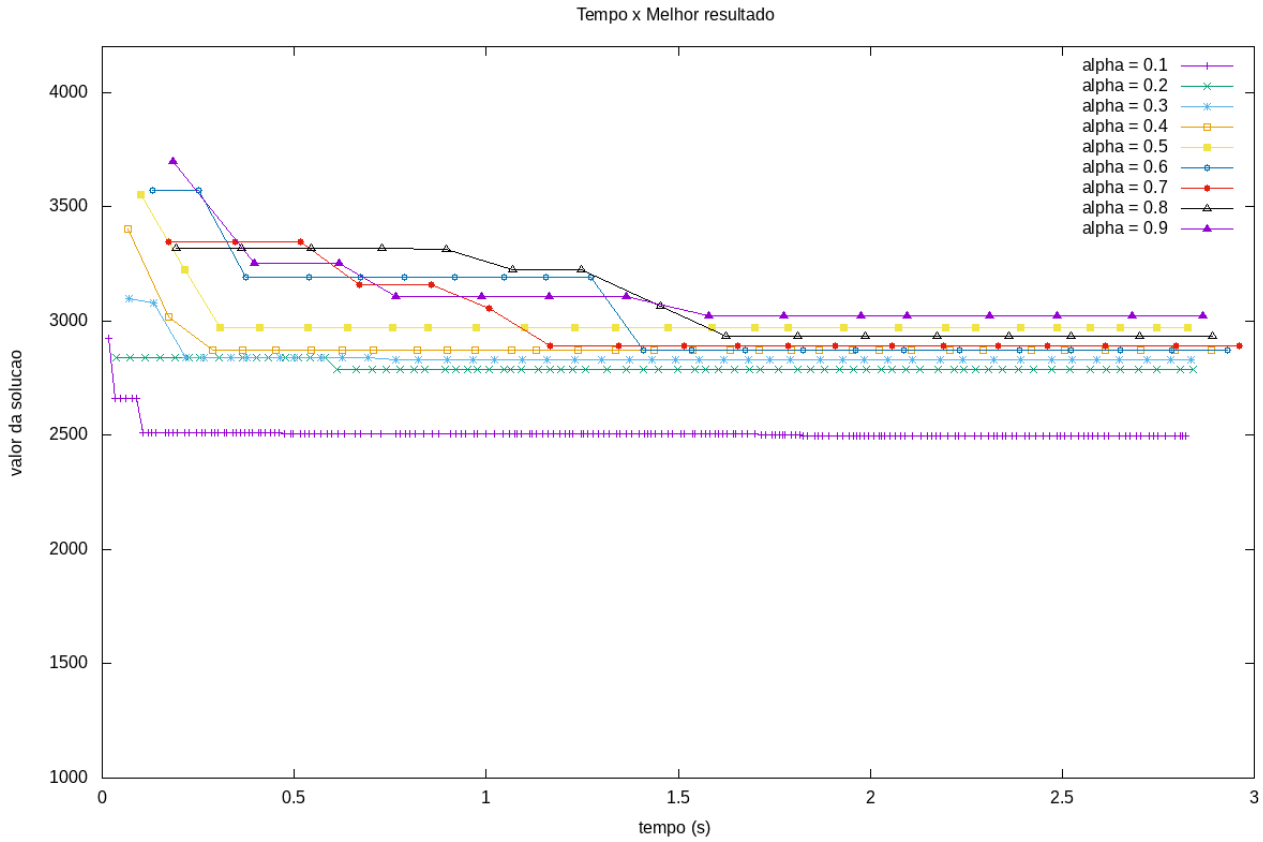

Figura 6: Melhor solução para a instância ftv70

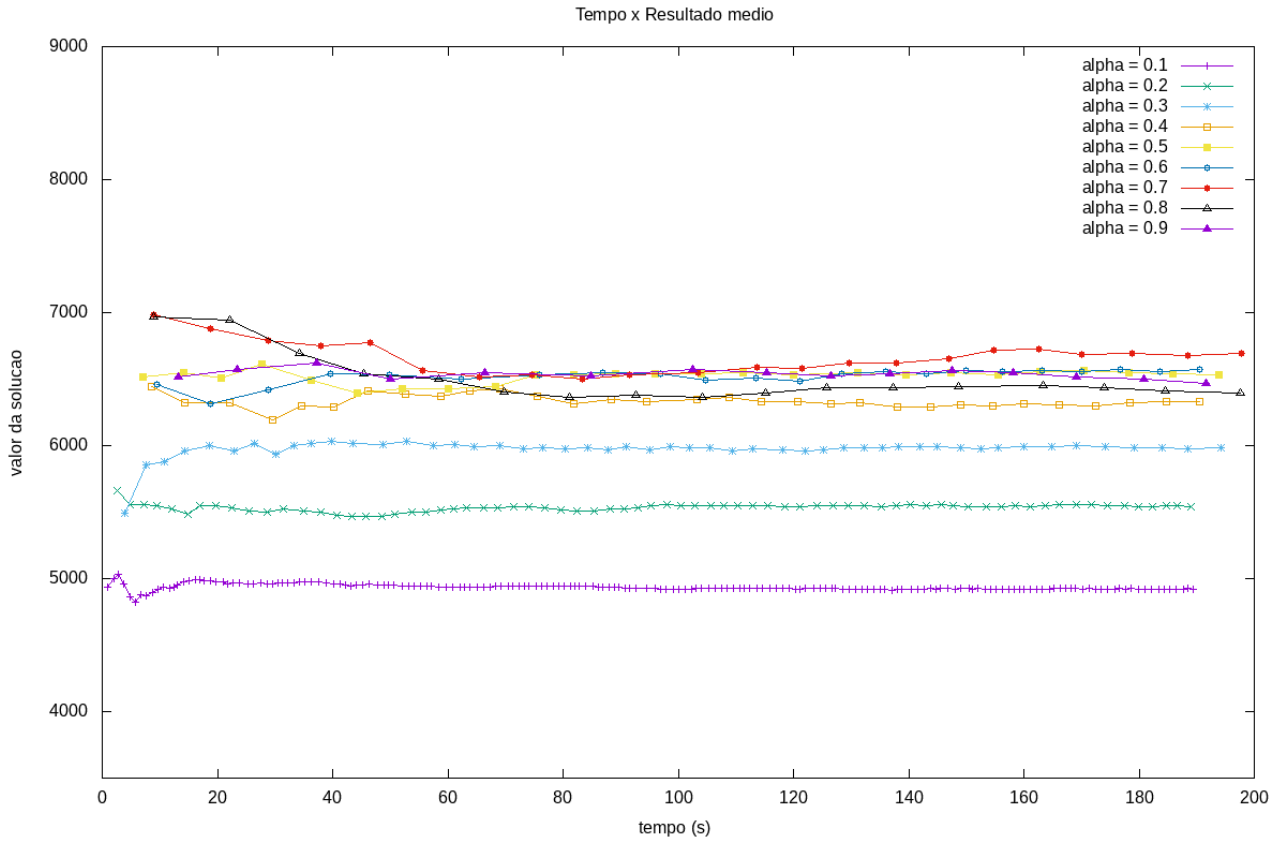

Figura 7: Solução média para a instância ftv170

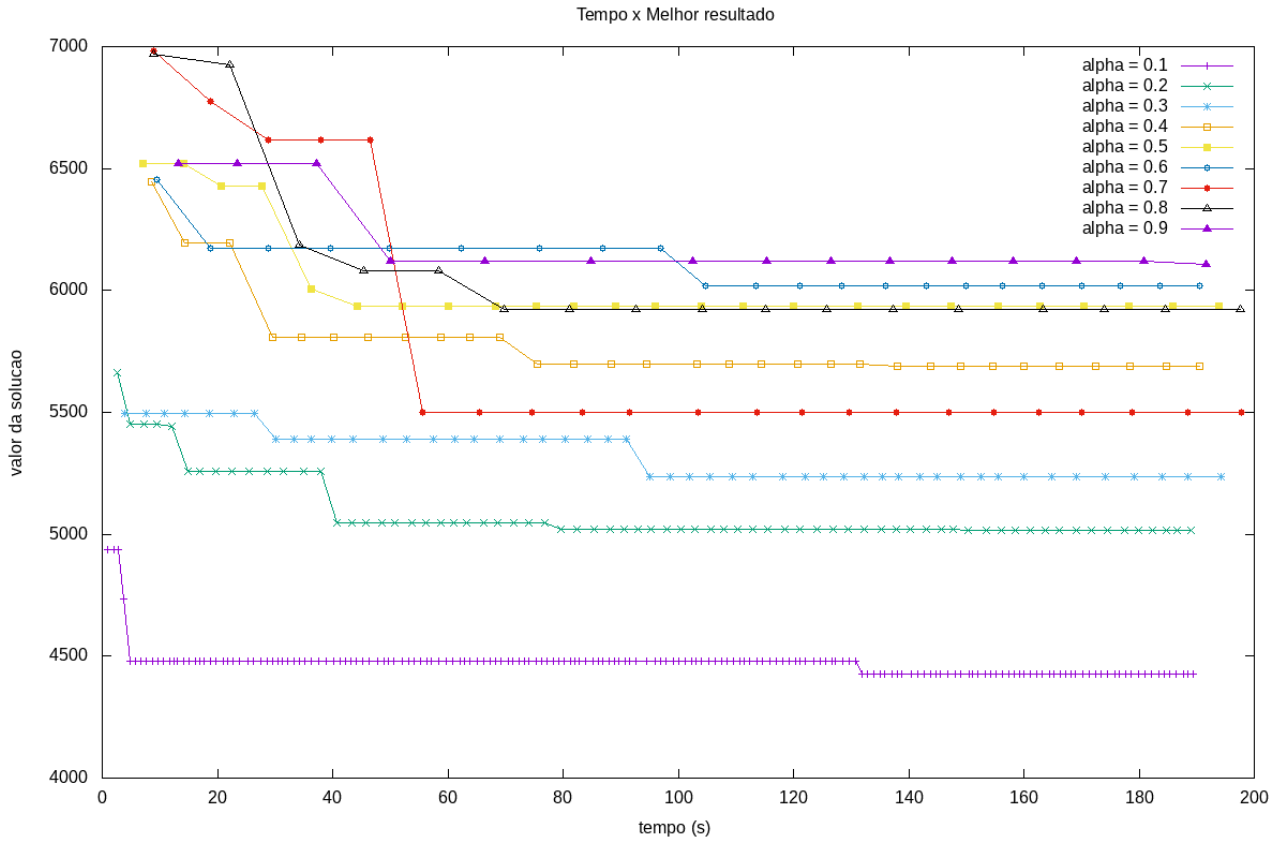

Figura 8: Melhor solução para a instância ftv170

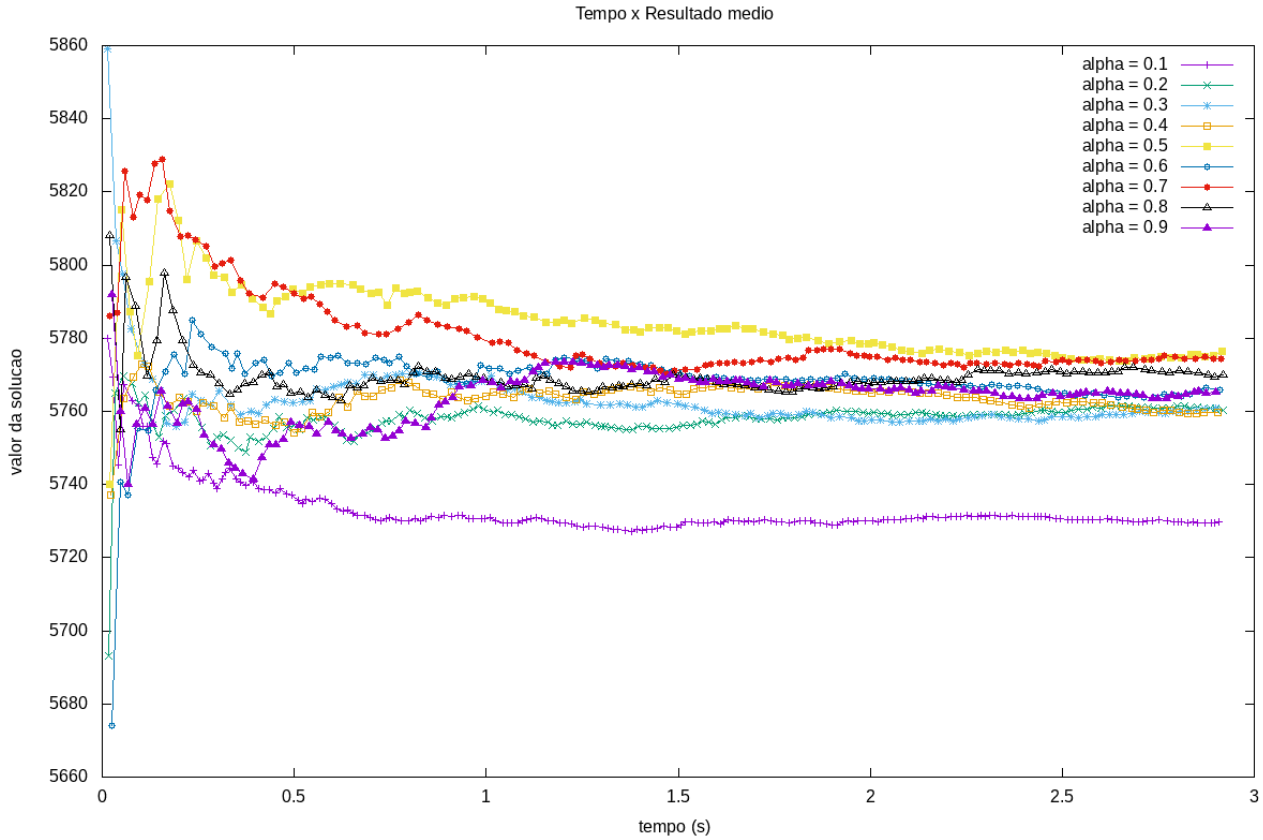

Figura 9: Solução média para a instância p43

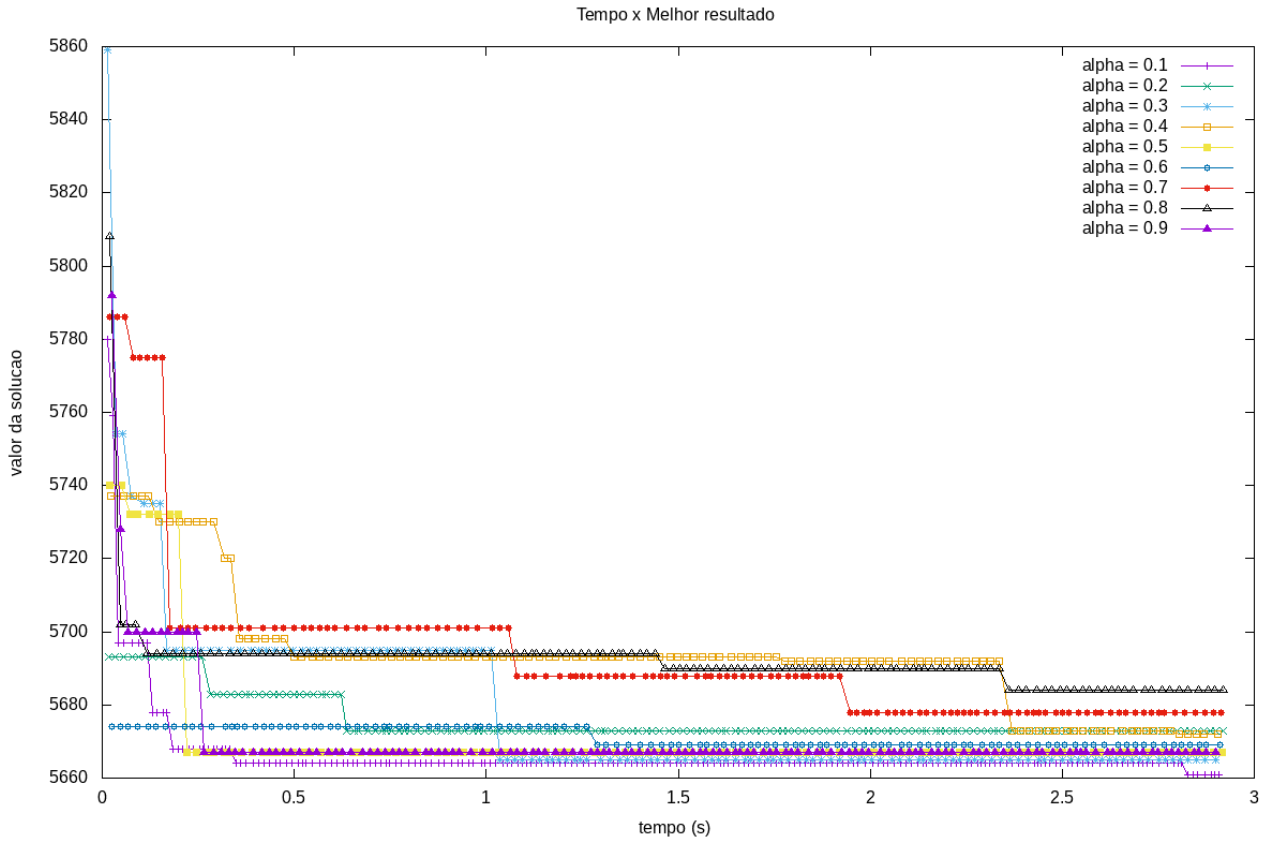

Figura 10: Melhor solução para a instância p43

Após selecionar alpha = 0.1 como padrão, foram executadas 1000 iterações do GRASP, para quatro instâncias, utilizando a estratégia primeira aprimorante. O tempo da execução foi capturado e utilizado para executar novamente o GRASP com estratégia melhor aprimorante. Para isso, foram escolhidas instâncias com dimensão cujo tamanho não fosse muito grande.
Como pode ser visto nas figuras abaixo (Figura 11 à Figura 14), o GRASP sempre obteve resultados melhores que a Busca Local em alguma das estratégias. Pode também ser visto que a estratégia primeira aprimorante leva vantagem sobre a estratégia melhor aprimorante em quase todos os casos, considerando o melhor resultado para o tempo de execução estabelecido. Como no experimento da implementação 2, novamente, a estratégia primeira aprimorante obteve melhores resultados.

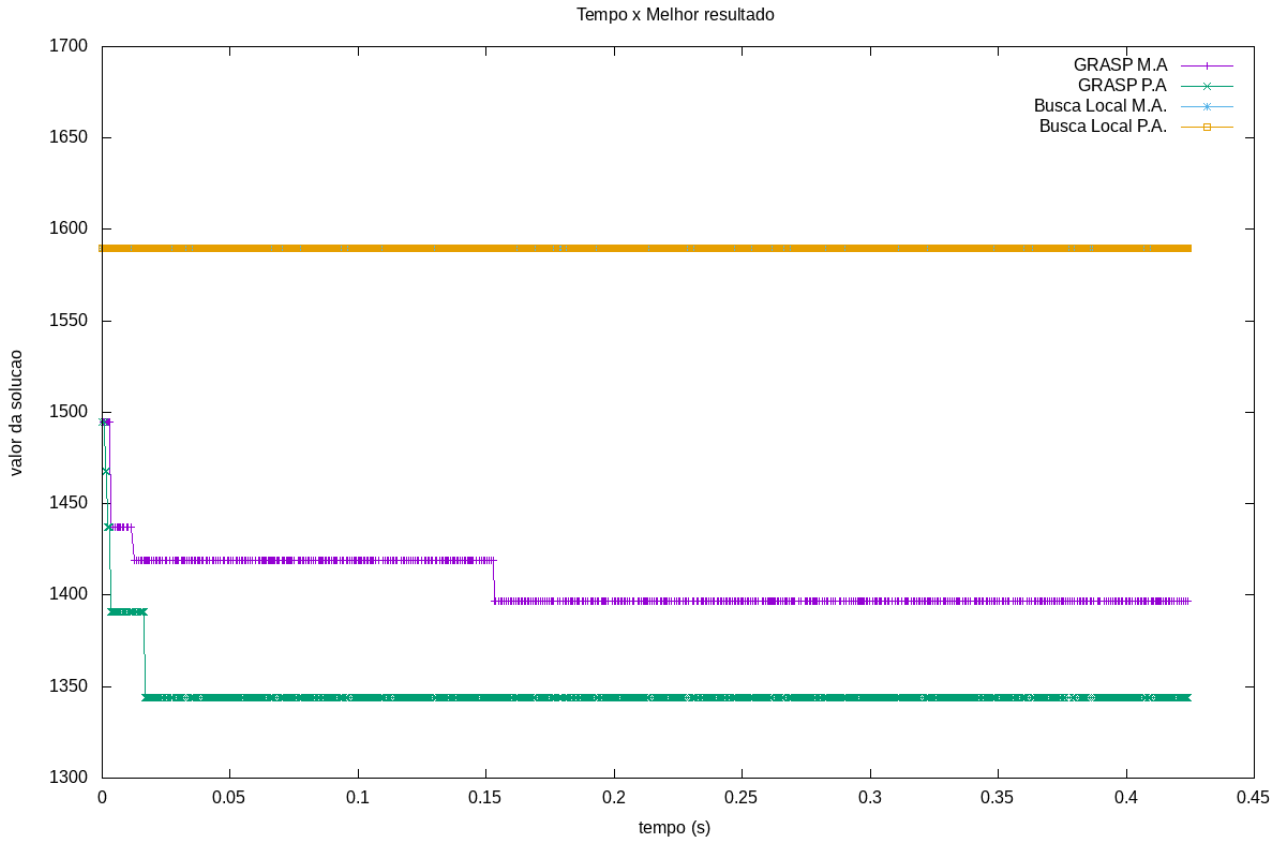

Figura 11: Comparações para a instância ftv33

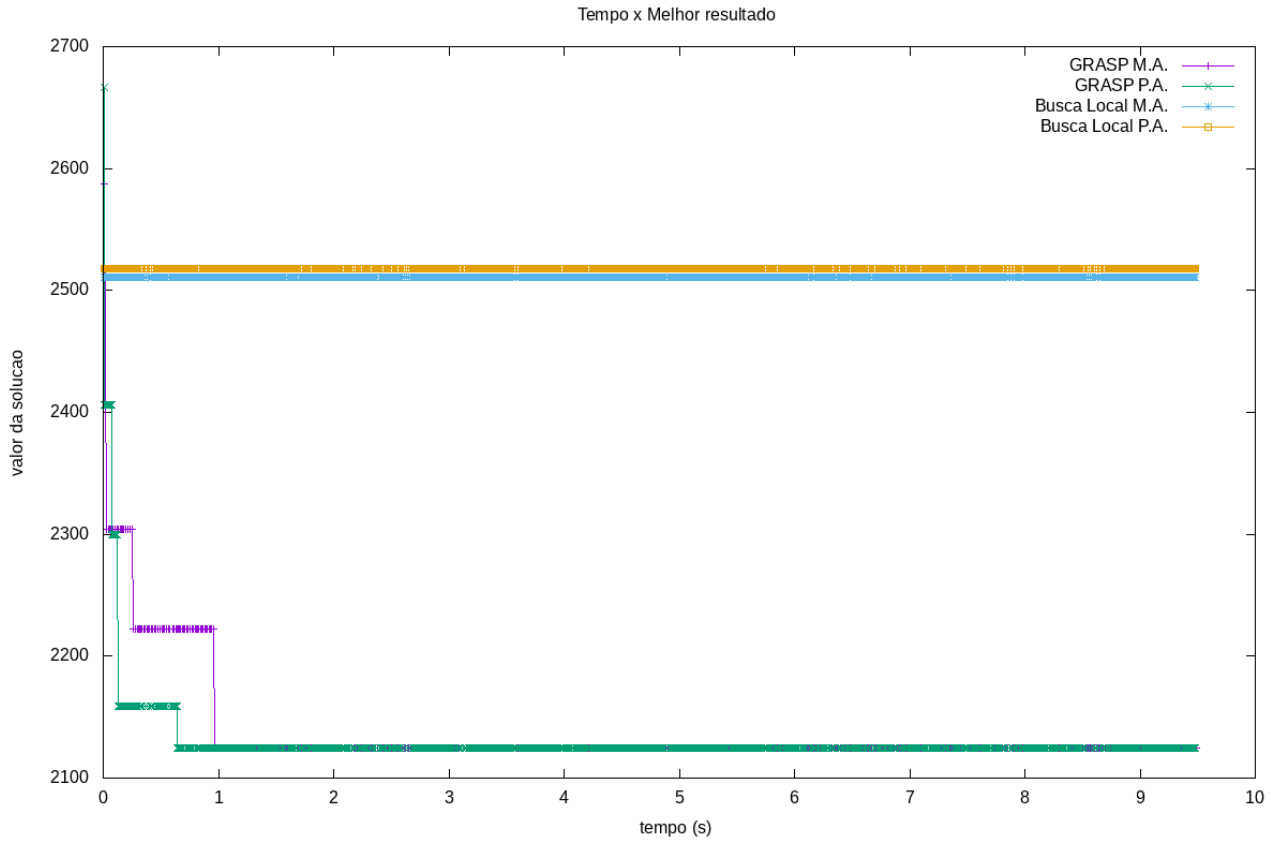

Figura 12: Comparações para a instância ftv64

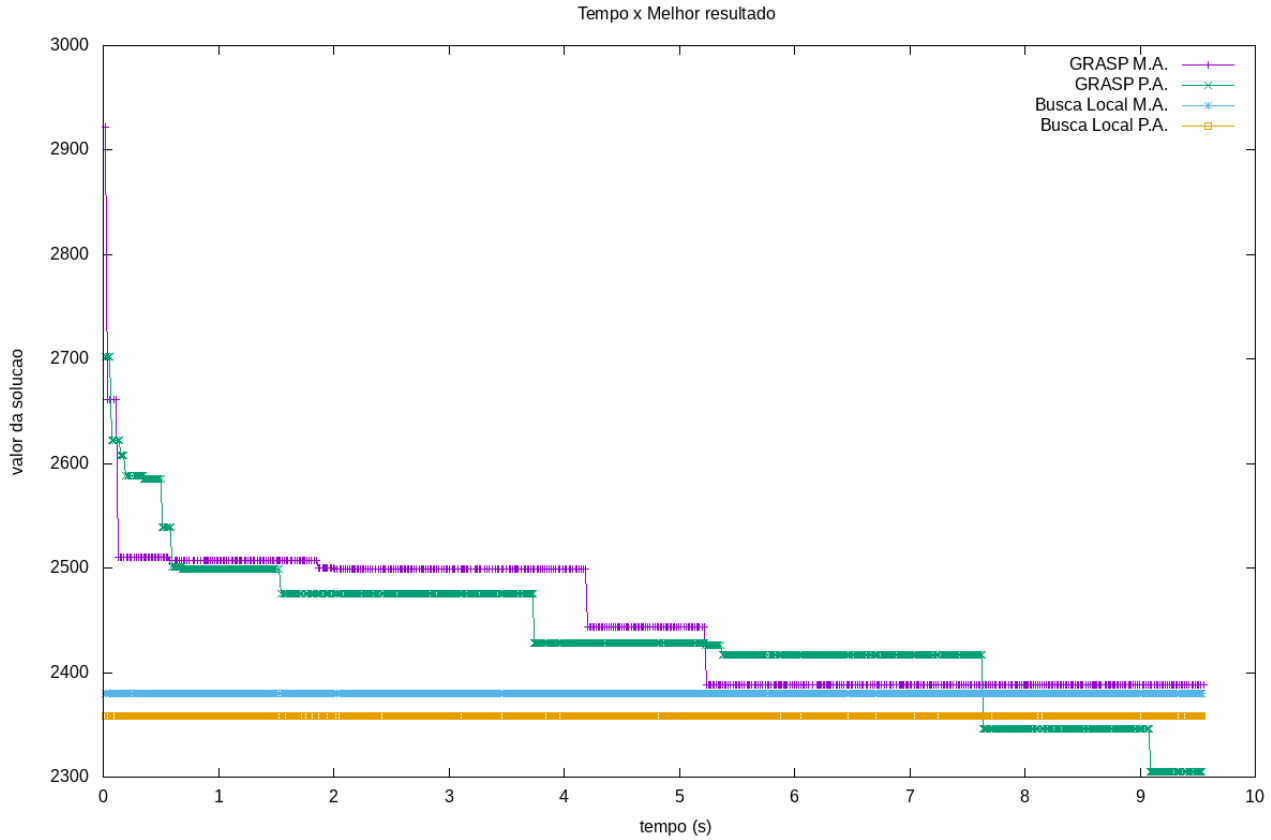

Figura 13: Comparações para a instância ftv70

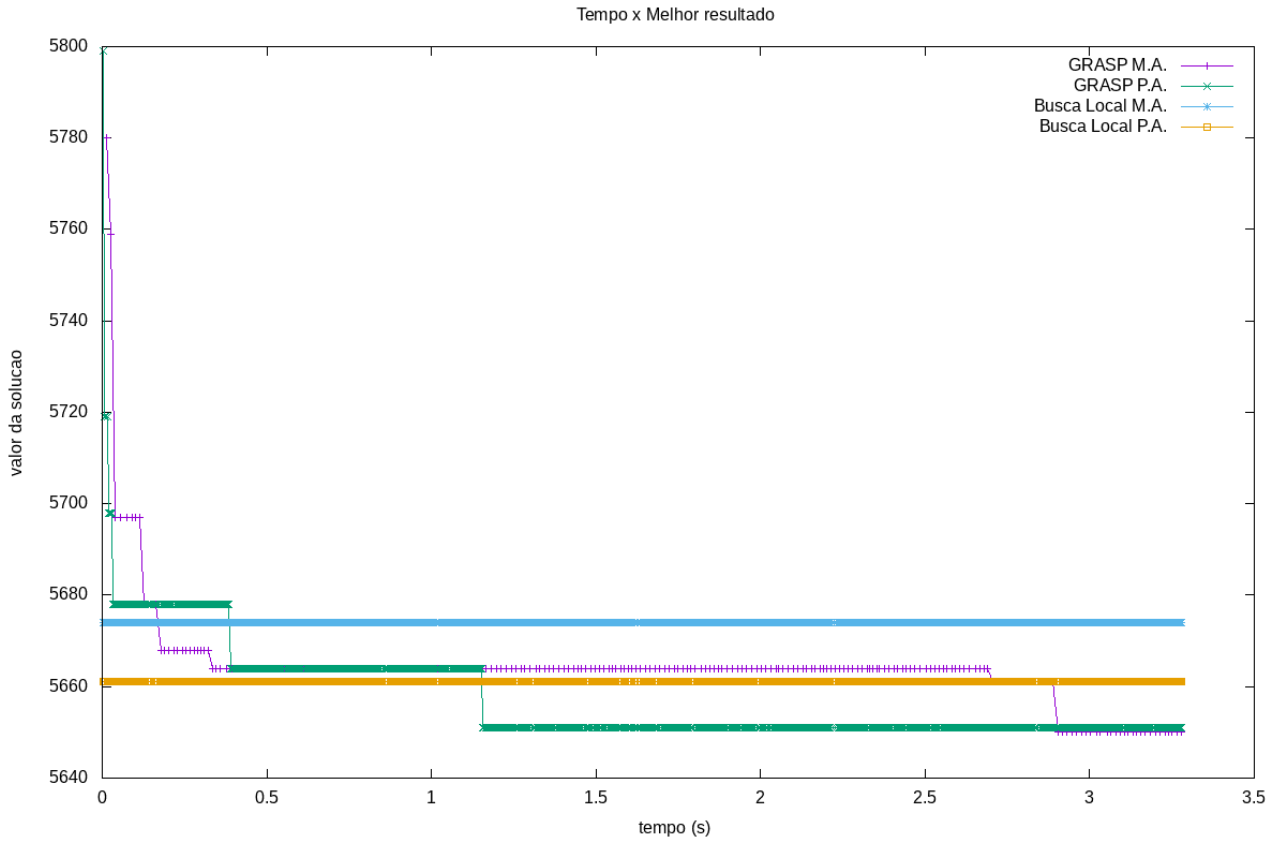

Figura 14: Comparações para a instância p43

Como podemos ver na tabela abaixo, o GRASP nos permite reduzir o valor da solução na maior parte dos casos, porém não sempre, dado que ele é um método que utiliza randomização. Para a estratégia mais aprimorante, apenas os resultados da instância ftv70 foram piorados, ao invés de melhorados. Para a estratégia primeira aprimorante, apenas os resultados da instância p43 foram piorados. Além disso, como podemos ver, essas piores e deram em um grau pequeno, beirando 0%.

Instância	ftv33	p43	ftv64	ftv70
Busca L. MA	1590	5674	2511	2380
Busca L. PA	1590	5661	2518	2359
GRASP MA	1397	5650	2125	2388
GRASP PA	1344	5664	2125	2305
Redução MA	12,14%	0,42%	15,37%	-0,34%
Redução PA	15,47%	-0,05%	15,61%	2,29%

Tabela 1: Redução do resultado de acordo com as políticas aprimorantes utilizadas em função do GRASP.

Quarta implementação

O quarto trabalho consistiu na implementação do método GRASP integrado com a técnica de Path Relink. Para isso, algumas definições foram feitas primeiramente:

- A estratégia a ser utilizada é a Mais Aprimorante.
- Para comparar o tempo entre execuções da mesma instância com o mesmo algoritmo, serão utilizadas várias sementes diferentes no algoritmo de randomização.
- Será utilizado a versão Backward do Path Relink.
- Para garantir uma maior variedade de soluções, será utilizado alpha = 0.2 ao invés de 0.1.

Após essas definições, o algoritmo foi implementado e os testes foram feitos em cinco instâncias. Para obter os tempos de diversas execuções de uma mesma instância com um mesmo algoritmo, foi utilizado um script SHELL como o script abaixo:

```
gcc -o exe grasp_pr.c

for i in {1..100}
 do
 ./exe files/ftv170.atsp i >> ftv170_6000_grasp.dat
 done
```

Este script realiza a compilação do código referente ao GRASP com Path Relink, e, após isso, executa 100 vezes o arquivo compilado, utilizando em todas as vezes o arquivo **ftv170.atp** e enviando a variável **i** (que varia de 1 até 100) como segundo parâmetro. O código, por sua vez, está preparado para abrir o arquivo especificado no primeiro parâmetro e utilizar o segundo parâmetro como semente para a randomização, garantindo que cada uma das execuções será diferente.

Um script como esse foi utilizado tanto para o GRASP com Path Relink quanto para o GRASP orininal (cujo alpha também foi ajustado para 0.2) e as comparações a seguir foram feitas através da comparação desses resultados.

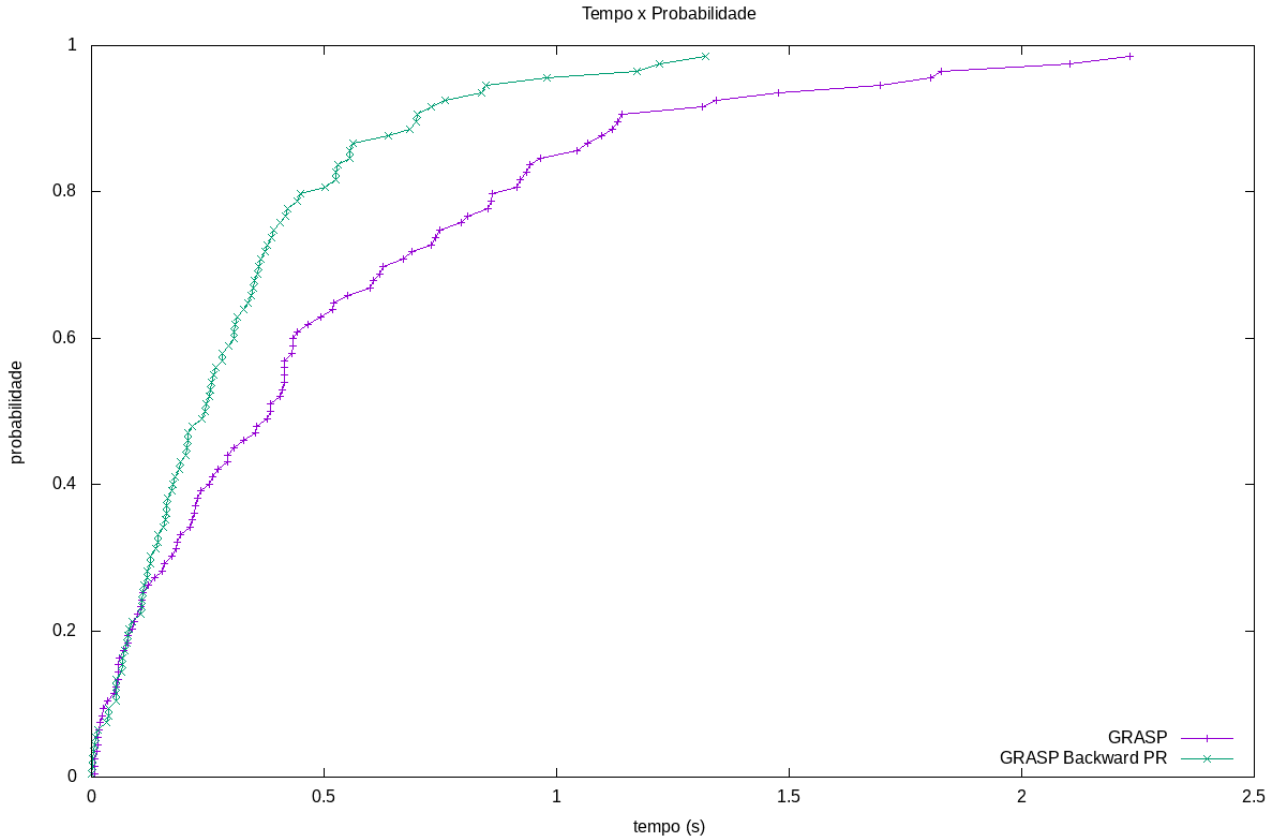

Figura 1: Instância ftv33

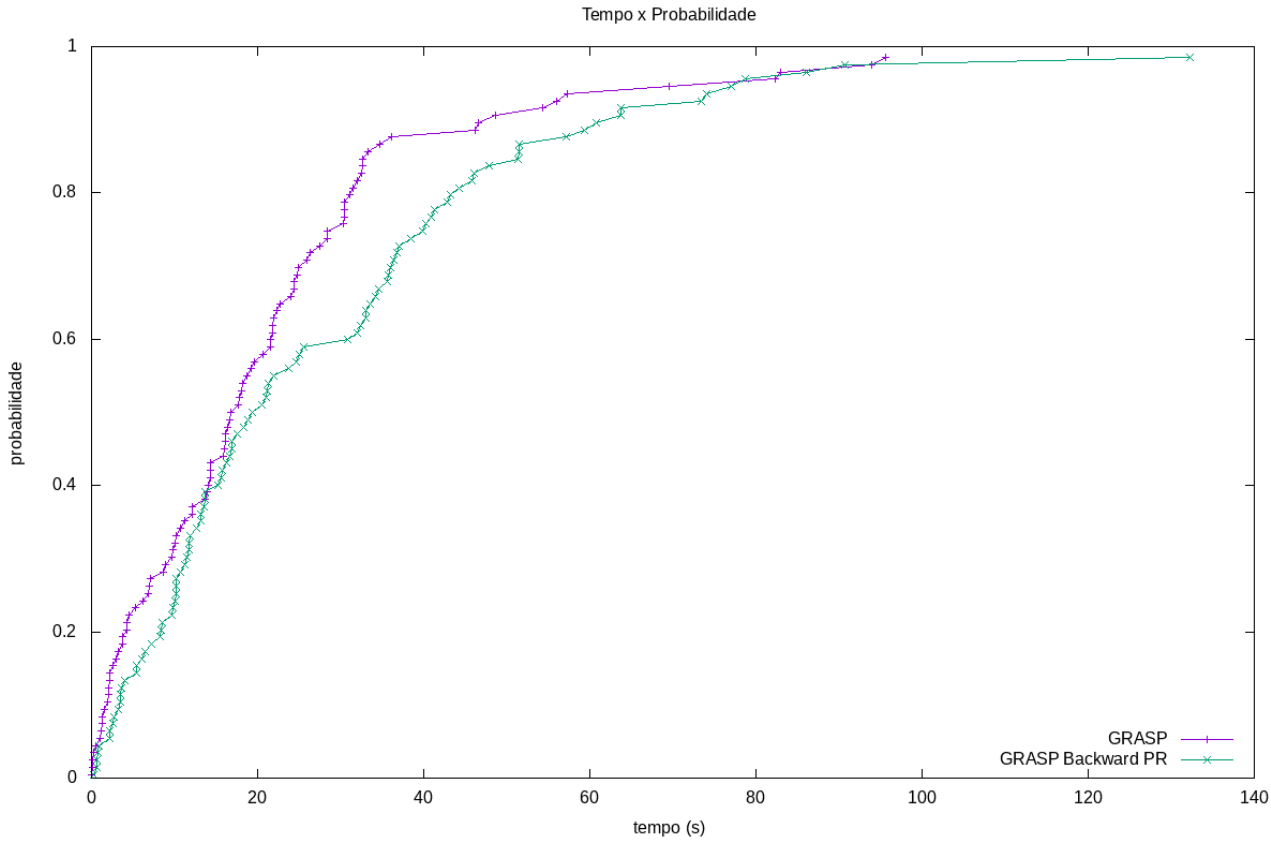

Figura 2: Instância ftv64

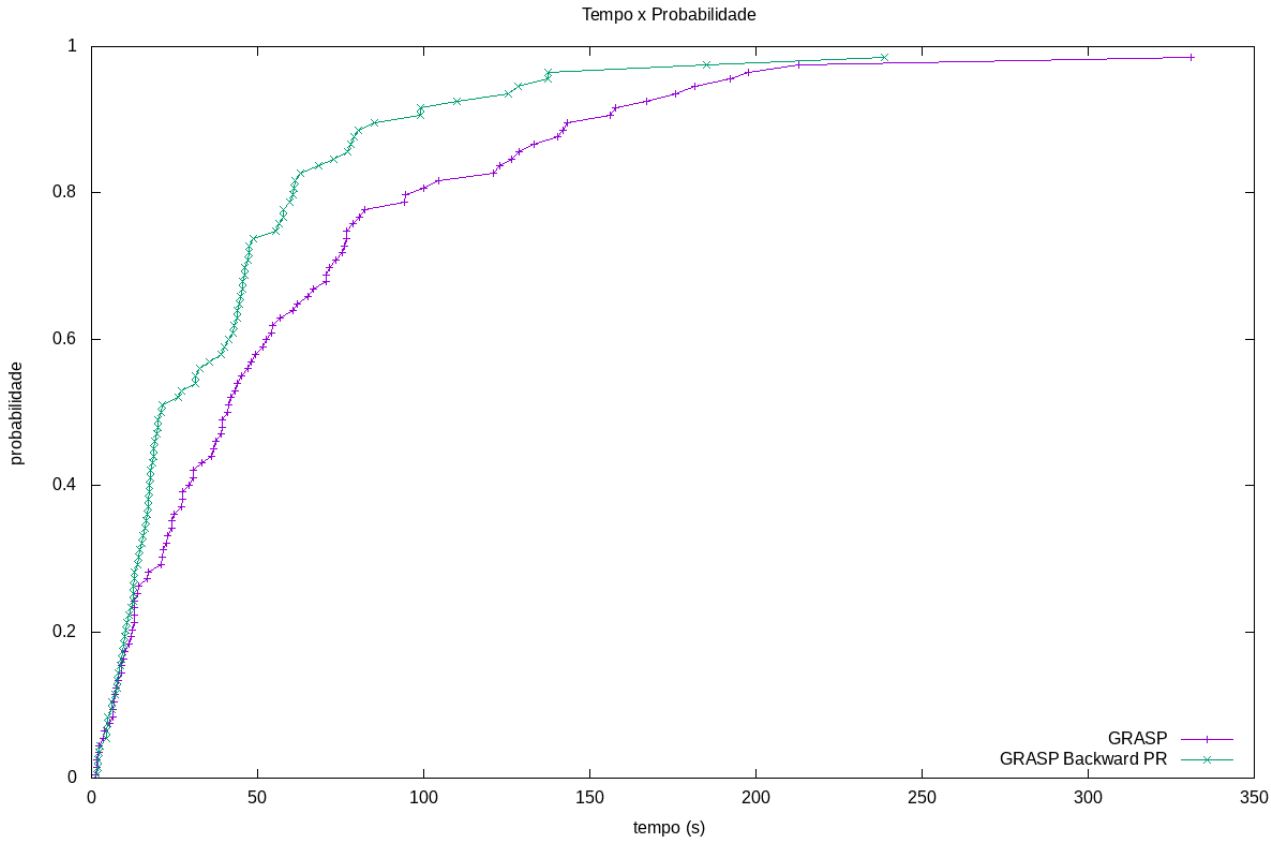

Figura 3: Instância ftv70

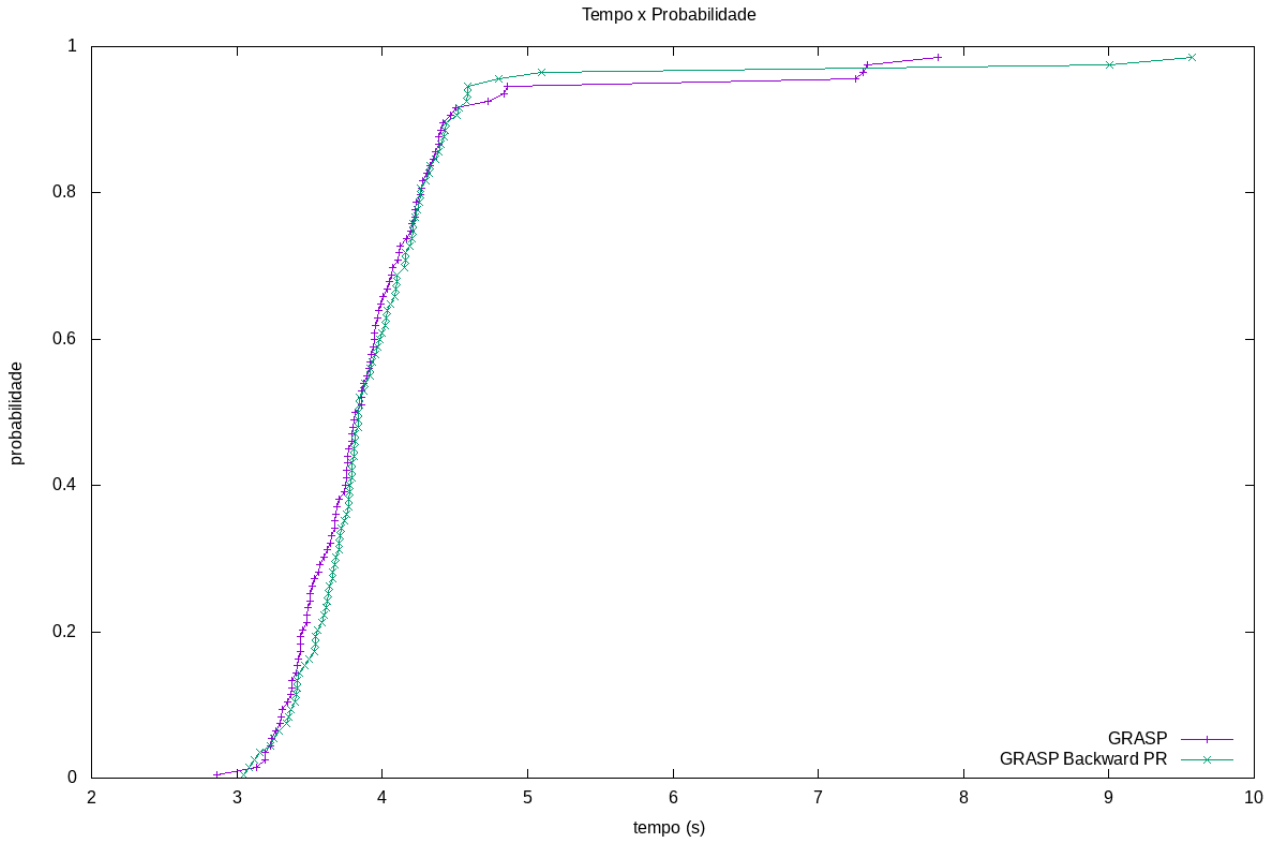

Figura 4: Instância ftv170

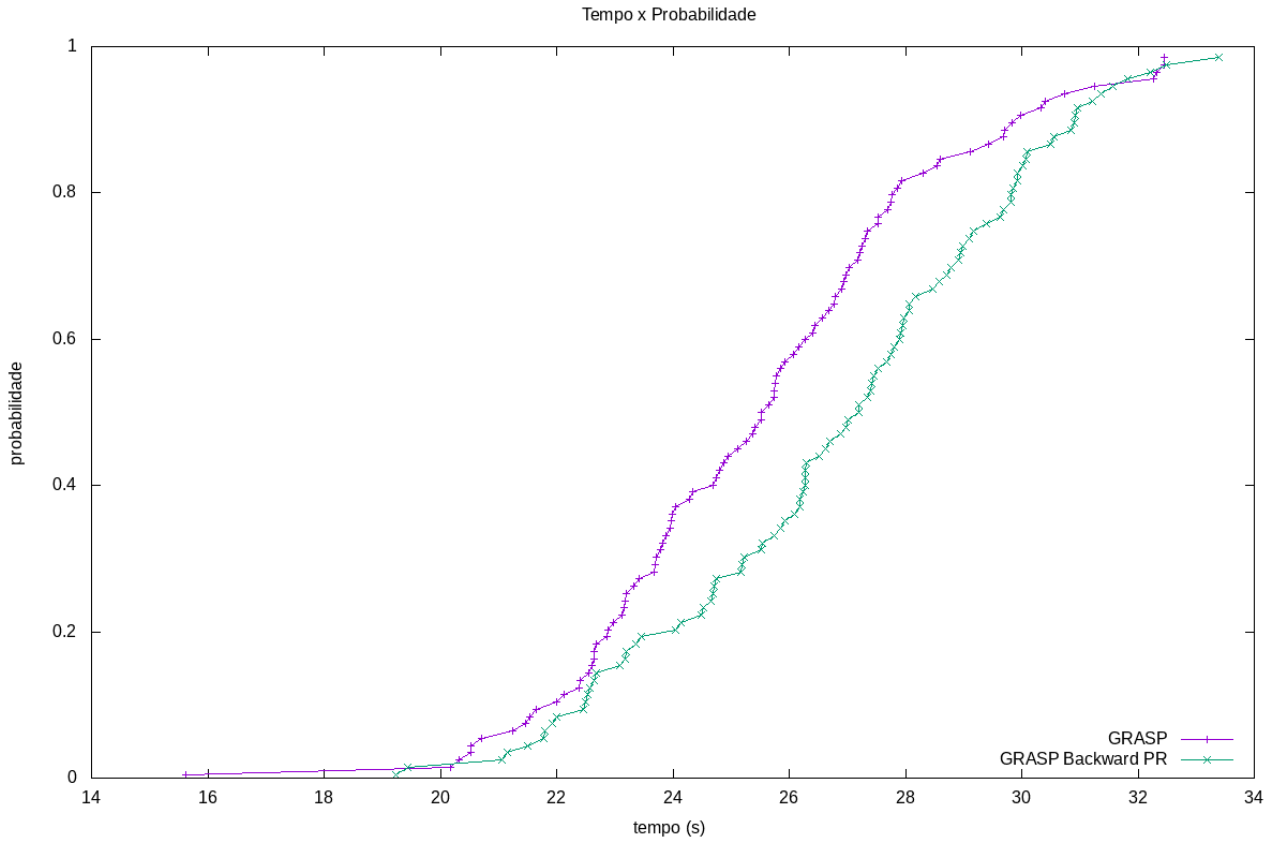

Figura 5: Instância rbg323

Curiosamente, como pode ser visto nas figuras, o algoritmo que utiliza Path Relink foi em média igual ao algoritmo que não utiliza Path Relink. Na maioria dos casos o GRASP original atingiu o alvo mais rápido do que o GRASP com Path Relink, o que se apresenta como uma anomalia.

O algoritmo ainda exige uma devida investigação para concluir o porque dessa anomalia (O algoritmo GRASP já era bom o suficiente? O algoritmo de Path Relink não ficou bom como deveria? Os problemas utilizados não são bons para o algoritmo em questão?)