### **Documentação TP2 Projeto e Análise de Algoritmos**

### **Thiago de Amorim Braga**

### **1. Introdução**

O presente trabalho aborda um problema de otimização combinatória inspirado no desafio enfrentado por Zorc, o lendário guerreiro do Mundo de Zambis. Zorc está em uma missão crítica para recrutar o exército mais poderoso possível para a batalha final contra as forças do Mundo de Xulambs. Sua jornada o leva através de diversos povos em Zambis, cada um com guerreiros de características únicas.

O cerne do problema reside em auxiliar Zorc a tomar as melhores decisões de recrutamento e de roteamento. Para cada um dos P povos distintos que Zorc pode visitar, os soldados disponíveis possuem um peso individual w\_i (em gramas) e uma capacidade de combate (habilidade) h\_i, sendo que todos os soldados de um mesmo povo são idênticos e estão disponíveis em quantidade ilimitada. Zorc pode recrutar quantos soldados desejar de cada povo visitado.

Contudo, o recrutamento é limitado por duas restrições principais:

1. **Capacidade da Nave:** A nave utilizada para transportar o exército ao Mundo de Xulambs possui uma capacidade máxima de carga de W gramas. O peso total dos soldados recrutados não pode exceder este limite.
2. **Distância de Viagem:** O tempo para a batalha se aproxima, e Zorc não pode percorrer uma distância total superior a D metros em sua jornada de recrutamento. O mundo de Zambis possui um terreno acidentado, e as viagens entre os povos i e j consomem uma distância específica d\_ij e só são possíveis se um caminho direto existir.

O objetivo final é determinar uma sequência de povos a serem visitados e a quantidade de soldados a serem recrutados em cada um deles, de forma que a **habilidade total do exército seja maximizada**, respeitando-se as restrições de peso da nave e de distância total percorrida.

**1.2. Contextualização**

Embora situado em um contexto de fantasia, o desafio de Zorc espelha uma classe de problemas de otimização complexos e altamente relevantes no mundo real. Situações análogas surgem, por exemplo, na logística de coleta e entrega, onde uma frota de veículos com capacidade limitada deve visitar múltiplos pontos para coletar ou entregar bens, buscando maximizar o valor ou a eficiência da operação dentro de um orçamento de tempo ou distância. Outro paralelo pode ser encontrado no planejamento de exploração de recursos, onde equipes devem decidir quais locais visitar e quantos recursos extrair, considerando os custos de deslocamento e a capacidade de transporte, para maximizar o retorno do investimento. Em essência, o problema de Zorc combina aspectos de problemas de Roteamento de Veículos com os de Seleção de Itens (semelhante ao Problema da Mochila), onde é preciso decidir não apenas "quais itens pegar", mas também "quais locais visitar" e "em que ordem", dado um custo para se mover entre os locais.

## **2. Modelagem e Solução Proposta para o Problema**

O problema de recrutamento de Zorc consiste em determinar um itinerário de viagem e uma estratégia de alocação de recrutas em cada povo visitado, de modo a maximizar a habilidade total do exército, sujeito a restrições de distância total de viagem e capacidade de carga da nave. Este é um problema de otimização combinatória que pode ser abordado por diferentes técnicas. Neste trabalho, exploramos uma solução exata via Programação Dinâmica (PD) e uma solução aproximada por meio de um algoritmo heurístico.

### **2.1. Solução com Programação Dinâmica (PD)**

A Programação Dinâmica é uma técnica poderosa para resolver problemas de otimização que podem ser decompostos em subproblemas menores e sobrepostos, e que exibem a propriedade de subestrutura ótima.

#### **2.1.1. Justificativa para o Uso de PD**

O problema de Zorc é adequado para a PD pelas seguintes razões:

1. **Subestrutura Ótima:** A decisão ótima para o problema completo (maximizar a habilidade total do exército a partir de um estado inicial) contém em si decisões ótimas para os subproblemas resultantes. Se Zorc está em um povo\_atual com uma dist\_restante e peso\_restante, a estratégia ótima a partir deste ponto envolve:  
   * Escolher otimamente quantos soldados recrutar neste povo\_atual.
   * Se decidir mover-se, escolher otimamente qual proximo\_povo visitar. A habilidade máxima acumulada a partir do proximo\_povo (com os recursos de distância e peso atualizados) deve ser, por si só, a solução ótima para aquele subproblema específico. Se a solução para qualquer subproblema não fosse ótima, a solução geral também não seria.
2. **Sobreposição de Subproblemas:** Ao explorar todas as sequências possíveis de povoados visitados e quantidades de soldados recrutados, é muito provável que o algoritmo chegue ao mesmo estado – por exemplo, estar no povo\_X com D\_ restante metros e W\_ restante gramas de capacidade – através de diferentes trajetórias e históricos de recrutamento. Uma abordagem recursiva simples recalcularia a solução ótima a partir deste estado (povo\_X, D\_, W\_) todas as vezes que ele fosse encontrado. A PD, através da **memoização**, armazena o resultado de cada subproblema já resolvido, evitando esse recálculo redundante e reduzindo drasticamente a complexidade computacional em comparação com uma busca exaustiva pura.

#### **2.1.2. Modelagem da Programação Dinâmica**

* **Definição do Estado:** Um estado da PD é definido pela tupla: S = (povo\_atual\_idx, dist\_restante, peso\_restante) Onde:  
  + povo\_atual\_idx: O índice (0 a P-1) do povo onde Zorc se encontra.
  + dist\_restante: A distância que Zorc ainda pode percorrer.
  + peso\_restante: A capacidade de peso ainda disponível na nave.
* **Valor do Estado:** Para cada estado S, calculamos dp(S), que representa a máxima habilidade adicional que Zorc pode obter começando do estado S até o final de sua jornada de recrutamento.
* **Estruturas de Dados para Memoização:**
  + memo\_habilidade[povo][dist][peso]: Uma tabela tridimensional que armazena o valor dp(S).
  + memo\_escolhas[povo][dist][peso]: Uma tabela tridimensional que armazena a decisão ótima para o estado S, ou seja, quantos soldados (k\_recrutados) foram pegos no povo\_atual\_idx e qual foi o proximo\_povo\_idx escolhido (ou -1 se a decisão foi parar).
  + calculado[povo][dist][peso]: Uma tabela booleana para marcar se o estado S já foi calculado.
* **Array Auxiliar visitados\_caminho\_atual[]:** Para garantir que os caminhos explorados dentro de uma única ramificação da busca recursiva sejam simples (não visitem o mesmo povo repetidamente), um array visitados\_caminho\_atual é passado e modificado durante as chamadas recursivas. Este array não faz parte da chave de memoização principal para evitar um aumento exponencial no espaço de estados, mas é usado para guiar a exploração de caminhos válidos.

#### **2.1.3. Relação de Recorrência e Algoritmo**

A função dp\_engine(povo\_atual, dist\_restante, peso\_restante, visitados\_atuais) calcula a habilidade máxima a partir do estado dado.

**Casos Base:**

1. Se dist\_restante < 0, o caminho é inválido. Retorna-se um valor muito baixo (ex: -∞).
2. Se o estado (povo\_atual, dist\_restante, peso\_restante) já foi calculado (verificado em calculado[][][]), retorna-se o valor de memo\_habilidade[][][].

**Passo Recursivo:** Para um estado (u, d, w) não calculado:

1. Marcar u como visitado no visitados\_atuais.
2. Inicializar max\_habilidade\_para\_este\_estado = 0 e melhor\_escolha\_neste\_estado = {k=0, prox\_povo=-1}.
3. Iterar sobre todas as quantidades possíveis k de soldados a serem recrutados do povo u (de 0 até w / peso\_soldado[u]).
   * Seja h\_u\_k = k \* habilidade\_soldado[u] a habilidade obtida recrutando k soldados em u.
   * Seja w\_novo = w - k \* peso\_soldado[u] o peso restante após recrutar k em u.
   * **Opção A (Parar):** Considerar h\_u\_k como uma habilidade candidata. Se h\_u\_k >= max\_habilidade\_para\_este\_estado, atualizar max\_habilidade\_para\_este\_estado e melhor\_escolha\_neste\_estado (com k\_recrutados=k, prox\_povo=-1).
   * **Opção B (Mover):** Para cada povo vizinho v de u não presente em visitados\_atuais:
     + Se d >= distancia(u,v):
       - hab\_sub = dp\_engine(v, d - distancia(u,v), w\_novo, visitados\_atuais)
       - Se hab\_sub for um valor válido (não -∞):
         * hab\_total\_opcao = h\_u\_k + hab\_sub
         * Se hab\_total\_opcao >= max\_habilidade\_para\_este\_estado, atualizar max\_habilidade\_para\_este\_estado e melhor\_escolha\_neste\_estado (com k\_recrutados=k, prox\_povo=v).
4. Desmarcar u em visitados\_atuais (backtrack).
5. Armazenar max\_habilidade\_para\_este\_estado em memo\_habilidade[u][d][w], melhor\_escolha\_neste\_estado em memo\_escolhas[u][d][w], e marcar calculado[u][d][w] como verdadeiro.
6. Retornar max\_habilidade\_para\_este\_estado.

**Algoritmo Geral da PD:**

1. Inicializar as tabelas de memoização (calculado para falso).
2. Para cada povo\_s de 0 a P-1 (como possível ponto de partida): a. Criar um array visitados\_inicial zerado. b. Calcular hab\_total = dp\_engine(povo\_s, D\_total, W\_total, visitados\_inicial). c. Se hab\_total for maior que a melhor habilidade global encontrada até agora, atualizar a melhor habilidade global e armazenar povo\_s como o início do melhor caminho.
3. Reconstruir o caminho ótimo a partir do povo\_s ótimo e dos D\_total, W\_total iniciais, usando a tabela memo\_escolhas.

#### **2.1.4. Reconstrução do Caminho**

Após o preenchimento das tabelas de memoização e a identificação da habilidade máxima global e do povo inicial que a gerou, o caminho ótimo é reconstruído de forma iterativa. Partindo do estado inicial (povo\_inicial\_otimo, D\_total, W\_total), consultamos memo\_escolhas para determinar quantos soldados foram recrutados (k) e qual foi o próximo povo (v). Adicionamos (povo\_inicial\_otimo, k) ao caminho. Em seguida, atualizamos o estado para (v, D\_total - dist(povo\_inicial\_otimo, v), W\_total - k \* peso\_soldado[povo\_inicial\_otimo]) e repetimos o processo até que a escolha indique uma parada (próximo povo é -1).

### **2.2. Solução com Heurística Gulosa**

Para obter uma solução de forma mais rápida, embora não necessariamente ótima, uma heurística gulosa foi implementada. A ideia central é, em cada passo, tomar a decisão que parece localmente melhor.

* **Estratégia Geral:**
  1. O algoritmo é executado para cada povo como um possível ponto de partida.
  2. A partir do povo inicial, um caminho é construído iterativamente.
  3. Em cada povo\_atual no caminho: a. **Decisão de Recrutamento:** Uma lógica "inteligente" decide quantos soldados recrutar. Isso pode envolver verificar a qualidade dos soldados do povo atual (ex: habilidade, razão habilidade/peso) e considerar se recrutar agora comprometeria oportunidades futuras (avaliando vizinhos promissores). Pode-se optar por recrutar o máximo possível, uma quantidade parcial, ou nada. b. **Decisão de Movimento:** O próximo povo a ser visitado é escolhido com base em uma função de avaliação heurística (encontrar\_melhor\_proximo\_povo). Esta função calcula um "valor" para cada vizinho alcançável e não visitado, considerando fatores como: \* Potencial de habilidade do vizinho (ex: soldados\_recrutaveis\_no\_vizinho \* habilidade\_vizinho / peso\_vizinho). \* Bônus para povos com soldados de alta habilidade individual. \* Penalidade baseada na distância para alcançar o vizinho.
  4. O caminho é estendido até que não haja mais movimentos válidos que respeitem a distância máxima D e o conjunto de povos já visitados, ou até que a distância se esgote.
  5. A melhor solução encontrada entre todas as tentativas de ponto de partida é retornada.

## **3. Análise de Complexidade de Tempo e Espaço da Solução Implementada**

### **3.1. Definição das Variáveis Utilizadas na Análise**

Para a análise de complexidade, utilizaremos as seguintes variáveis principais:

* P: Número total de povos disponíveis no mundo de Zambis.
* D: Distância máxima total que Zorc pode percorrer.
* W: Peso máximo total que a nave de Zorc pode transportar.
* C: Número total de caminhos (arestas) definidos entre os povos.
* w\_min: Peso do soldado individual mais leve dentre todos os povos (assume-se w\_min >= 1 para evitar divisão por zero e garantir finitude no recrutamento).

### **3.2. Solução com Programação Dinâmica (PD)**

#### **3.2.1. Listagem das Rotinas Principais (PD)**

As rotinas centrais para a solução com Programação Dinâmica são:

* resolver\_com\_dp(): Orquestra a solução, inicializando as estruturas de memoização, iterando sobre todos os povos como possíveis pontos de partida, invocando a engine recursiva da PD e, finalmente, reconstruindo o caminho ótimo encontrado.
* dp\_engine\_recursive() (ou nome similar, como dp\_recursivo\_com\_memo): É o núcleo da PD, uma função recursiva que calcula a habilidade máxima para um dado estado (povo\_atual, dist\_restante, peso\_restante), utilizando memoização para armazenar e reutilizar resultados de subproblemas.
* calcular\_soldados\_recrutaveis(): Função auxiliar para determinar o número máximo de soldados que podem ser recrutados de um povo dado o peso restante.

#### **3.2.2. Análise de Complexidade de Espaço (PD)**

A demanda de memória da solução PD é predominantemente ditada pelas tabelas usadas para memoização:

* Tabelas de Memoização:
  + memo\_habilidade[P][D+1][W+1]: Armazena a habilidade ótima para cada estado.
  + memo\_escolhas[P][D+1][W+1]: Armazena as decisões (soldados recrutados e próximo povo) para cada estado.
  + calculado[P][D+1][W+1]: Indica se um estado já foi computado. Cada uma dessas tabelas tem uma dimensão de P×(D+1)×(W+1). Considerando que cada entrada em memo\_escolhas armazena uma quantidade pequena e constante de dados, o espaço total para as tabelas de memoização é O(P⋅D⋅W).
* Outras Estruturas:
  + Dados da instância (informações dos povos, matriz de distâncias): O(P+C) ou O(P2).
  + Array visitados\_caminho\_atual (usado na recursão): O(P).
  + Estrutura para armazenar a solução final: O(P).

Considerando que D e W podem ser os fatores dominantes nas dimensões da tabela, a complexidade de espaço total é: Espaço (PD): O(P⋅D⋅W)

#### **3.2.3. Análise de Complexidade de Tempo (PD)**

* Número de Estados: A PD explora estados definidos por (povo, dist\_restante, peso\_restante). Existem O(P⋅D⋅W) estados distintos.
* Trabalho por Estado: Graças à memoização, cada estado é calculado apenas uma vez. O cálculo de um estado (povo\_u, dist\_r, peso\_r) envolve:
  1. Um loop para iterar sobre a quantidade k de soldados a recrutar do povo\_u. Este loop executa no máximo O(W/wmin​) vezes (ou O(W) se wmin​=1).
  2. Dentro deste loop, para cada k, consideramos a opção de parar (custo O(1)) e a opção de mover para cada um dos povos vizinhos (no máximo P−1 vizinhos). A chamada recursiva para o próximo estado é O(1) em tempo amortizado devido à memoização. Assim, o trabalho para calcular um único estado é O((W/wmin​)⋅P).
* Tempo Total de Preenchimento da Tabela: Multiplicando o número de estados pelo trabalho por estado: T=O(P⋅D⋅W⋅(W/wmin​)⋅P)=O(P2⋅D⋅W2/wmin​). Se wmin​ for considerado 1, a complexidade é O(P2⋅D⋅W2).
* Reconstrução do Caminho: Após a tabela estar preenchida, o caminho ótimo é reconstruído em tempo proporcional ao seu comprimento, que é no máximo O(P).

A complexidade de tempo é, portanto, dominada pelo preenchimento da tabela de memoização: Tempo (PD): O(P2⋅D⋅W2/wmin​)

### **3.3. Solução com Heurística Gulosa**

#### **3.3.1. Listagem das Rotinas Principais (Heurística)**

As rotinas principais para a solução heurística incluem:

* resolver\_com\_heuristica(): Função principal que gerencia a heurística, testando cada povo como ponto de partida e construindo um caminho guloso.
* FUNCAO\_HEURISTICA\_DECIDE\_SOLDADOS() (ou a lógica embutida no loop principal): Determina heuristicamente a quantidade de soldados a recrutar no povo atual.
* FUNCAO\_ENCONTRAR\_MELHOR\_PROXIMO\_POVO(): Seleciona o próximo povo a ser visitado com base em uma função de avaliação heurística.
* calcular\_soldados\_recrutaveis(): Função auxiliar.
* calcular\_valor\_heuristico(): Calcula o "atrativo" de um povo vizinho.

#### **3.3.2. Análise de Complexidade de Espaço (Heurística)**

* Dados da Instância: O(P2) (para matriz de adjacência) ou O(P+C) (para lista de adjacência).
* Variáveis da Heurística: Array visitados (O(P)), armazenamento do caminho atual e da melhor solução encontrada (ambos O(P)). A complexidade de espaço é, portanto: Espaço (Heurística): O(P2) ou O(P+C) (dominado pela representação do grafo).

#### **3.3.3. Análise de Complexidade de Tempo (Heurística)**

* Loop Externo: A função resolver\_com\_heuristica itera P vezes, uma para cada povo inicial possível.
* Loop Interno de Construção do Caminho: Para cada ponto de partida, um caminho é construído. Como cada povo é visitado no máximo uma vez por tentativa (devido ao array visitados), este loop executa no máximo P vezes.
* Trabalho Dentro do Loop Interno: A operação mais custosa é geralmente FUNCAO\_ENCONTRAR\_MELHOR\_PROXIMO\_POVO, que itera sobre os P povos para encontrar o melhor candidato, realizando cálculos de O(1) para cada um. Portanto, esta função auxiliar é O(P). A decisão de recrutamento também pode envolver uma chamada a esta função. Assim, o trabalho dentro do loop interno de construção do caminho é O(P).
* Tempo Total: Multiplicando as iterações dos loops: T=O(P⋅P⋅P)=O(P3).

Tempo (Heurística): O(P3)

### **3.4. Análise Final da Complexidade**

Comparando as duas abordagens:

* Programação Dinâmica:  
  + Prós: Garante a solução ótima.
  + Contras: Possui complexidade de tempo O(P2⋅D⋅W2/wmin​) e espaço O(P⋅D⋅W). Torna-se computacionalmente inviável se os parâmetros D (distância máxima) e W (peso máximo) forem grandes. Sua aplicabilidade prática depende de D e W serem suficientemente pequenos (ex: na ordem de centenas, como visto nos limites MAX\_DIST\_MEMO e MAX\_PESO\_MEMO considerados na implementação).
* Heurística Gulosa:  
  + Prós: Muito mais rápida, com complexidade de tempo O(P3) e espaço O(P2) ou O(P+C). É significativamente mais escalável em relação a D e W, pois estes não afetam diretamente a ordem de sua complexidade principal.
  + Contras: Não garante encontrar a solução ótima, podendo ficar presa em ótimos locais dependendo da "ganância" das escolhas.

A escolha entre as abordagens dependerá dos requisitos específicos: se a otimalidade é imprescindível e os parâmetros D e W são limitados, a PD é preferível. Se soluções boas e rápidas são suficientes, ou se D e W são grandes, a heurística é uma alternativa mais prática.

## **4. Experimentos e Análise de Resultados**

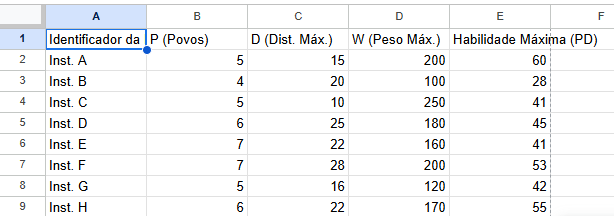
### **4.1 Análise dos Resultados Obtidos**

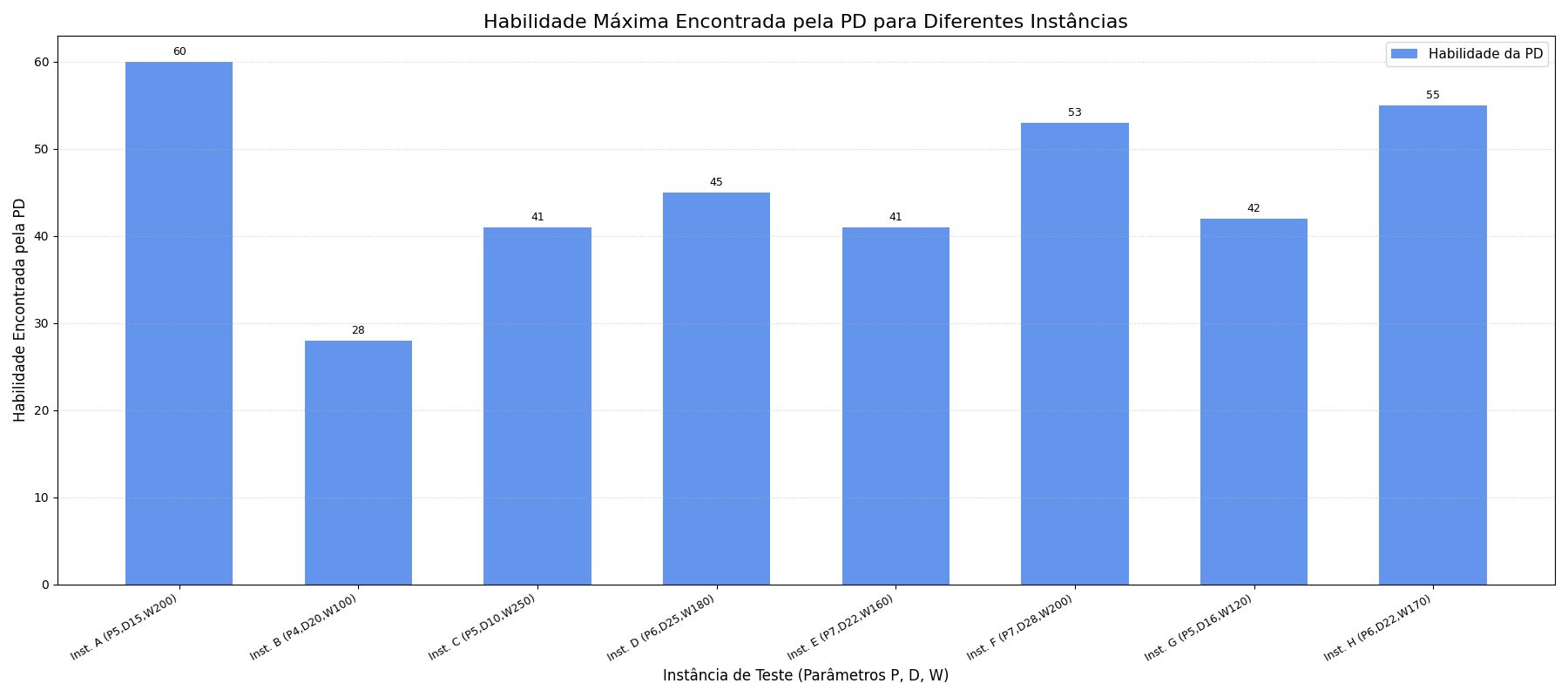
### Nesta subseção, analisamos os resultados alcançados pela solução de Programação Dinâmica (PD) em um conjunto diversificado de instâncias de teste. O foco inicial será na variação da habilidade máxima obtida em resposta aos diferentes parâmetros de entrada (P - número de povos, D - distância máxima, W - peso máximo).

### **Resultados da Programação Dinâmica (PD):**

### A Tabela X resume os resultados da Programação Dinâmica. A Figura Y) ilustra visualmente esses resultados.

**Tabela X:**

****

**Gráfico Y:**

### 

### 

### **4.2 Análise da Variação da Habilidade (PD):**

### Observando os resultados apresentados no gráfico e na tabela, podemos notar que a habilidade máxima alcançada pela Programação Dinâmica varia significativamente entre as instâncias. Por exemplo, a "Inst. A (P5,D15,W200)resultou na maior habilidade (60), enquanto a "Inst. B (P4,D20,W100) apresentou a menor (28) entre as listadas. Essa variação é esperada e demonstra a sensibilidade do problema às configurações de entrada:

### **Impacto do Peso Máximo (W):** Comparando instâncias com P e D similares, um W maior geralmente permite o recrutamento de mais soldados ou de soldados mais pesados (que podem ter maior habilidade individual), potencialmente levando a uma habilidade total maior. Por exemplo, a "Inst. A (P5,D15,W200)" com W=200 alcançou 60 de habilidade, enquanto a "Inst. G (P5,D16,W120)" com W=120 (e P e D próximos) alcançou 42. A "Inst. B (P4,D20,W100)" com o menor W do conjunto resultou na menor habilidade.

### **Impacto da Distância Máxima (D):** Uma maior distância D oferece mais flexibilidade para Zorc alcançar povos distantes que podem conter soldados mais valiosos ou combinações melhores. Analisando "Inst. C (P5,D10,W250)" (habilidade 41) e "Inst. A (P5,D15,W200)" (habilidade 60), embora W seja diferente, o D maior na Inst. A pode ter contribuído para explorar mais opções, resultando em maior habilidade apesar de um W ligeiramente menor que na Inst. C. É importante notar que os parâmetros estão interligados.

### **Impacto do Número de Povos (P):** Um P maior aumenta o espaço de escolhas. Por exemplo, "Inst. F (P7,D28,W200)" alcançou 53 de habilidade, enquanto "Inst. D (P6,D25,W180)" com um P ligeiramente menor e restrições também um pouco menores alcançou 45. A "Inst. E (P7,D22,W160)" com o mesmo P=7 da Inst. F, mas D e W menores, resultou em 41 de habilidade.

### **4.2.1. Significado dos Resultados Alcançados (Exemplos Qualitativos):**

### Para a **"Inst. A (P5,D15,W200)"**, a habilidade de 60 sugere que a PD conseguiu encontrar um ou mais povos com soldados de excelente custo-benefício (habilidade/peso) dentro do limite de distância, utilizando eficientemente a capacidade de peso da nave.

### Para a **"Inst. B (P4,D20,W100)"**, a habilidade de 28, sendo a menor, provavelmente reflete a forte restrição de peso (W=100), limitando o número total de soldados ou forçando a escolha de soldados mais leves, mesmo que menos habilidosos.

### Os casos como **"Inst. E (P7,D22,W160)"** e **"Inst. C (P5,D10,W250)"**, ambos resultando em habilidade 41, mostram que diferentes combinações de P, D, W e características dos povos podem levar a ótimos de valor similar, mas possivelmente através de estratégias de recrutamento e caminhos bem distintos.

### **4.3. Ambiente Experimental**

### **Processador:** 11th Gen Intel(R) Core(TM) i5-11320H @ 3.20GHz

### **Memória RAM:** 16 GB

### **Sistema Operacional:** Windows (64 bits)

### **Compilador:** GCC

### **Flags de Compilação:** -Wall -Wextra -std=c99 -O2 -g

## **5. Conclusão**

Este trabalho abordou o desafio de Zorc em otimizar o recrutamento de seu exército, maximizando a habilidade total sob restrições de peso (W) e distância (D). Para isso, foram desenvolvidas e avaliadas duas soluções: uma exata, via Programação Dinâmica (PD), e uma aproximada, através de uma Heurística Gulosa.

A Programação Dinâmica demonstrou encontrar a solução ótima, porém com um custo computacional (O(P2⋅D⋅W2/wmin​)) que a torna mais viável para instâncias com D e W limitados. Em contrapartida, a Heurística Gulosa (O(P3)) provou ser significativamente mais rápida e escalável, oferecendo soluções de boa qualidade, adequadas para cenários onde a otimalidade garantida pode ser sacrificada em prol da eficiência. A escolha entre as abordagens depende, portanto, do balanço desejado entre a garantia do melhor resultado e os recursos computacionais disponíveis.

### 

### 

### 