Relatório MAC0318 EP6

Renato Lui Geh — NUSP: 8536030

1 Mapa

O mapa criado para os testes foi gerado a partir de uma imagem 1024×720. Na imagem, há alguns polígonos rotacionados que criam os obstáculos. A área branca é desobstruída e as formas pretas são os obstáculos.

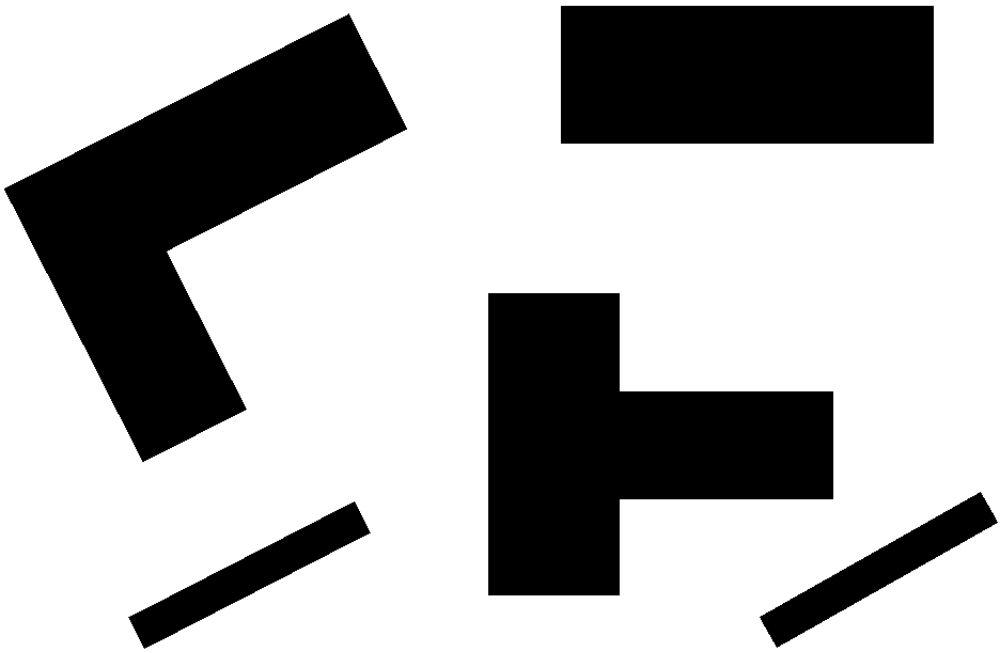

Figura 1: Mapa de polígonos onde a área branca indica desobstrução e formas pretas são obstáculos.

A classe `Map` representa o mapa de polígonos. Seu construtor aceita quatro argumentos: o *path* da imagem que representa o mapa, o tamanho do robô para dilatação, e o par (b_x, b_y) que parametrizam a discretização do mapa para o eixo x e y respectivamente.

Quando construído, `Map` gera dois mapas adicionais. O primeiro é a mesma imagem dilatada com o tamanho do robô. Para isso foi usada a função `binary_dilation` do pacote

`scipy.ndimage.morphology`. A estrutura binária usada para a dilatação foi uma matriz 3×3 com todos valores `true`. Desta forma, quando passada para a função de dilatação, será feita uma dilatação quadrada de tamanho d.

O segundo mapa é a discretização aplicada ao mapa dilatado. Para isso, foi considerada uma transformação de uma matriz $M(w, h)$ para $M(w/b_x, h/b_y)$, tal que toda região de pixel b_x por b_y é somada e normalizada entre 0 e 1 e em seguida transformada num só pixel $q = (q_x, q_y)$ cujo valor é o arredondamente da região original para o valor mais próximo 0 ou 1. Para isso, foi usada a função `resize` e `threshold` do pacote `cv2`.

2 Polígonos

Para se detectar os polígonos no mapa, foi usada a função `findContours` do pacote `cv2`. Esta função retorna os pontos de formas geométricas numa imagem. Estes pontos foram então juntados para formar as linhas que definem o polígono. Por causa que imagens são representadas por uma matriz, o espaço discreto faz com que haja imperfeições na detecção de formas. Então o número de linhas detectadas foi maior do que se a detecção fosse ideal. No total foram geradas 1038 linhas a partir desta função. A função também detecta a própria imagem como uma forma geométrica. Portanto, além dos polígonos interiores, foi gerada também as bordas da imagem.

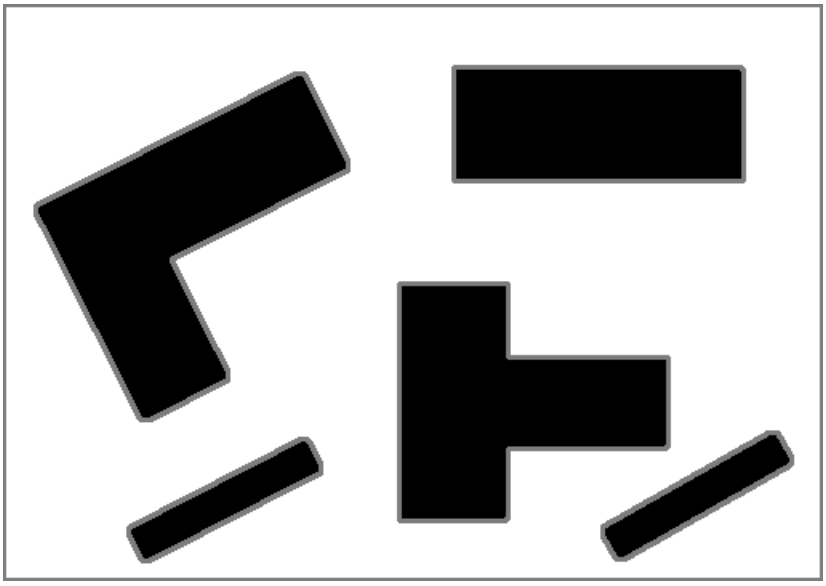

Figura 2: As linhas em cinza indicam a detecção das bordas das formas geométricas. Foram no total 1038 linhas geradas.

3 Configurações iniciais

Foram feitos três testes com pontos iniciais e finais diferentes. Vamos denotar por $p = (p_x, p_y)$ e $q = (q_x, q_y)$ os pontos iniciais e finais respectivamente.

Teste 1: $p = (10, 10)$, $q = (1014, 710)$

Teste 2: $p = (365, 61)$, $q = (365, 495)$

Teste 3: $p = (61, 365)$, $q = (495, 365)$

Teste 4: $p = (211, 555)$, $q = (588, 268)$

Teste 5: $p = (555, 211)$, $q = (268, 588)$

O primeiro teste tenta cobrir um caminho por toda a image. O segundo tenta testar o quão bem o algoritmo desvia do obstáculo. O terceiro é similar ao segundo, mas com os eixos trocados. O quarto é o percurso mais fácil, mas que percorre um caminho entre vários polígonos. No quinto, os eixos são trocados.

Para cada algoritmo, mostraremos uma imagem para cada teste feito, com o caminho feito pelo algoritmo dado por pontos cinza claros e o caminho após linearização dado por linhas cinza escuras.

Vamos nos referir ao Teste n como T.n. Então o Teste 3 será denotado por T.3.

4 Frente de onda

A função `wavefront` implementa o algoritmo, e a função `show_wavefront` cria uma imagem para visualização do caminho. O algoritmo foi razoávelmente rápido e teve bons resultados.

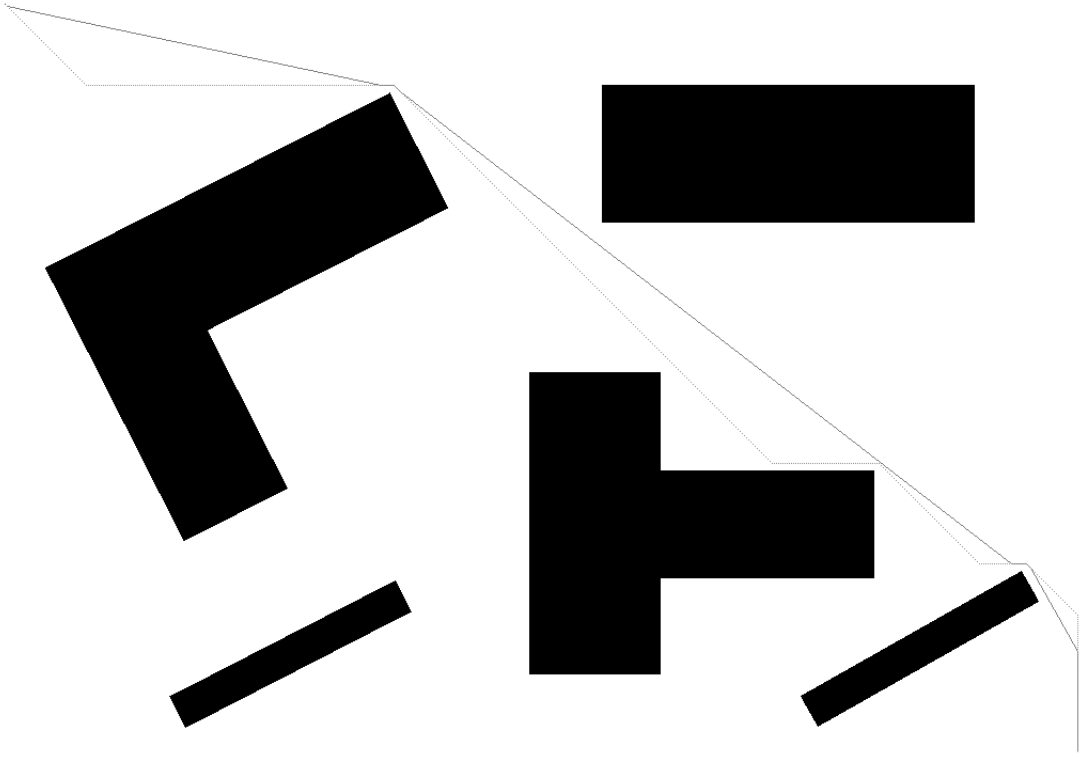

Figura 3: Caminho da frente de onda para o T.1.

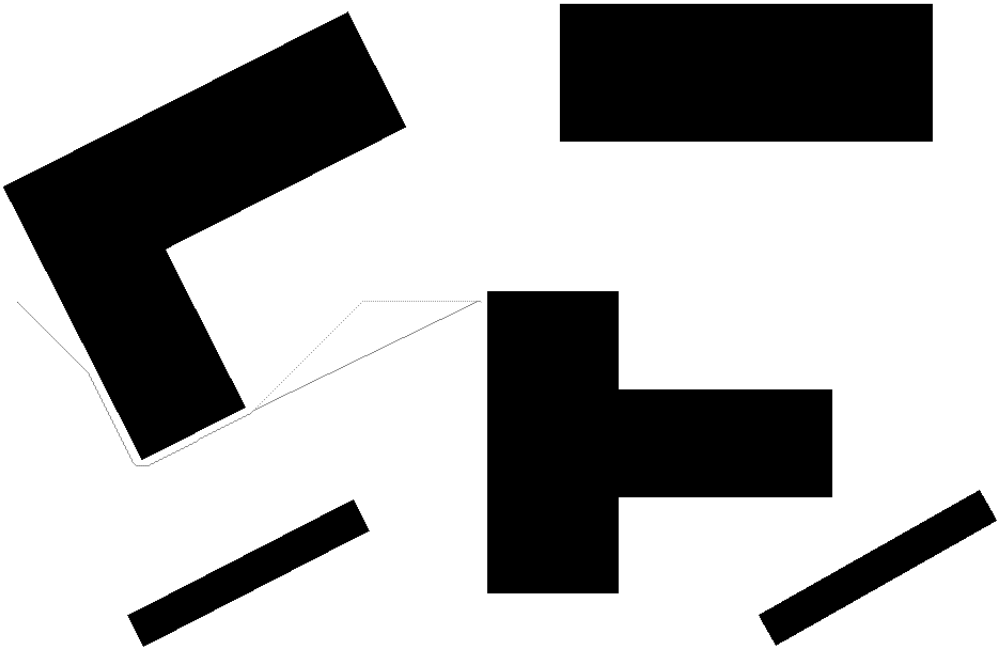

Figura 4: Caminho da frente de onda para o T.2.

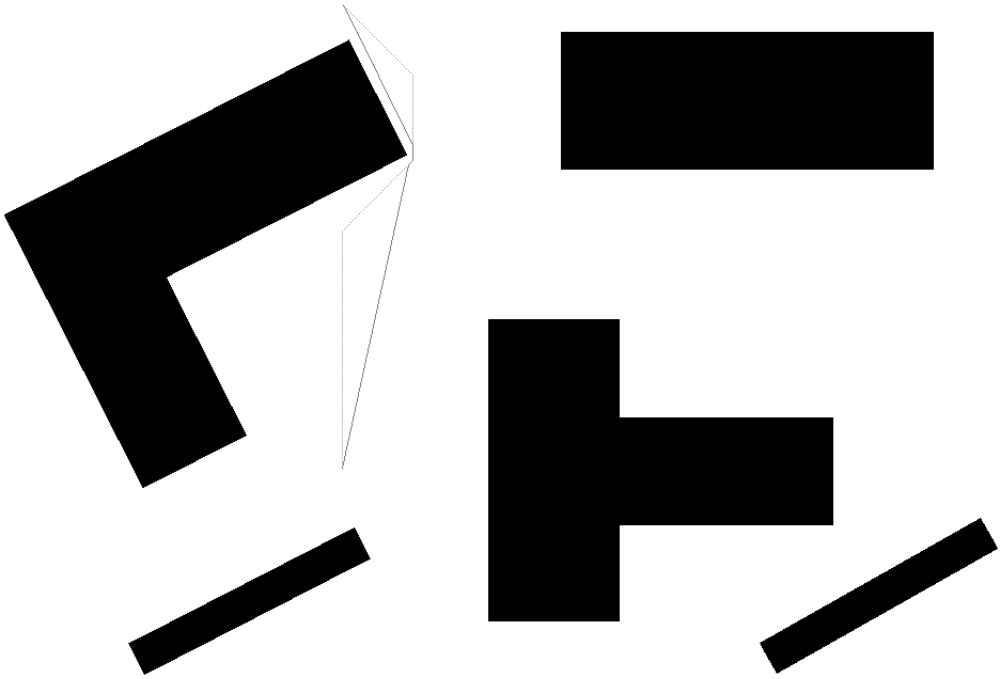

Figura 5: Caminho da frente de onda para o T.3.

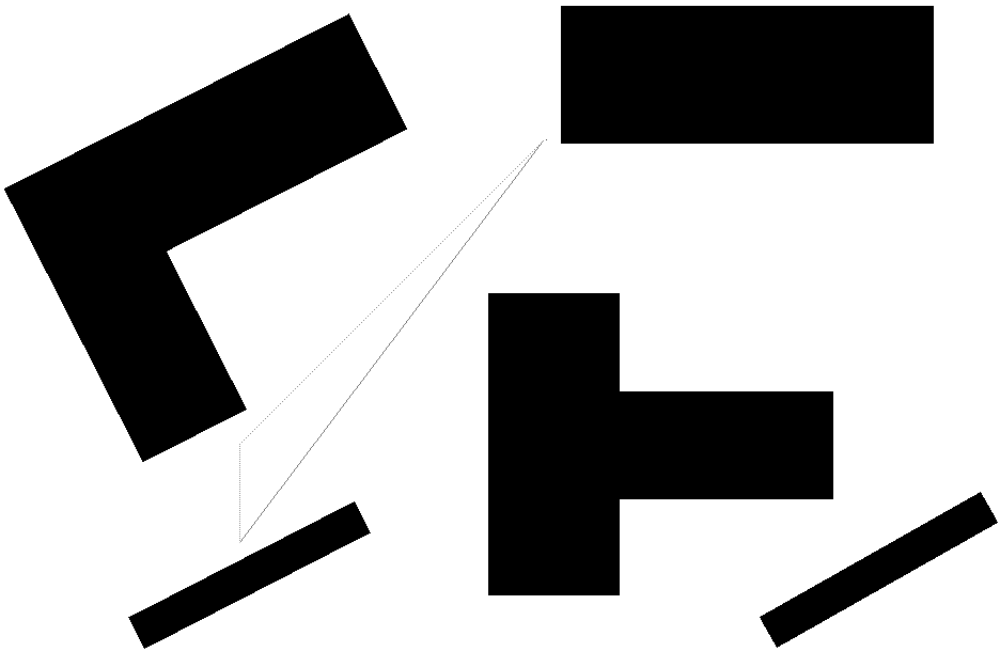

Figura 6: Caminho da frente de onda para o T.4.

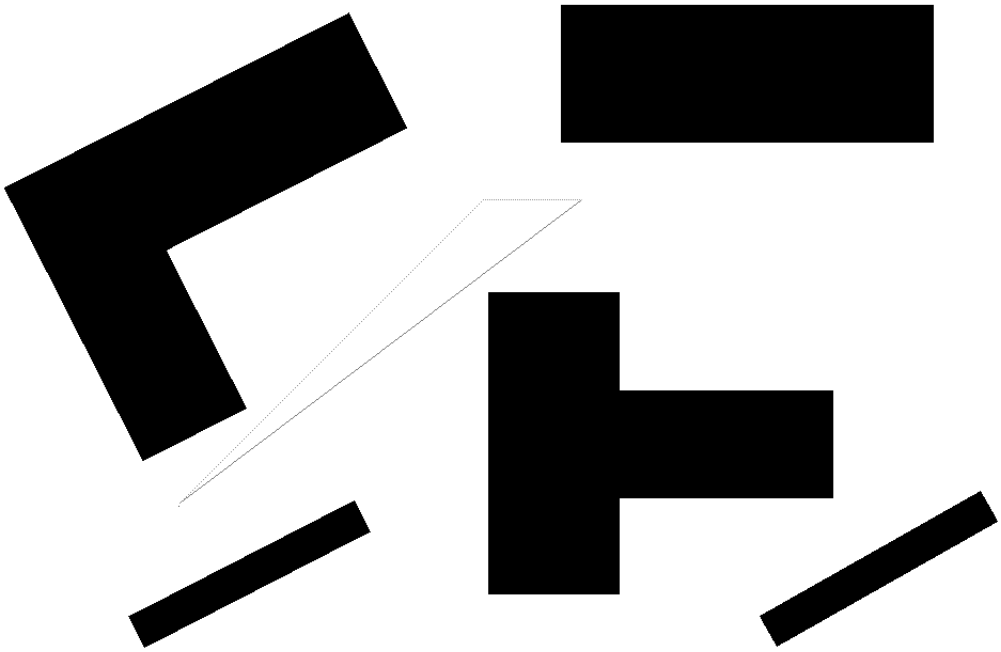

Figura 7: Caminho da frente de onda para o T.5.

5 Melhor escolha

O algoritmo de melhor escolha foi implementado pela função `best_choice`, com a função `show_best_choice` servindo para visualização do caminho.

Esta função toma um argumento adicional `mfunc`, que indica qual métrica usar: manhattan ou euclideana. A escolha deste parâmetro mudou drásticamente tanto os resultados quanto o quão rápido o algoritmo levou para computar os caminhos. A métrica Manhattan causou uma melhora muito significativa no tempo de execucação. Para todos os caminhos, levou-se tanto tempo quanto ou muito menos do que a frente de onda. No entanto, os resultados para o T.1 mostraram o problema que a distância de Manhattan gera por não considerar as diagonais menos custosas.

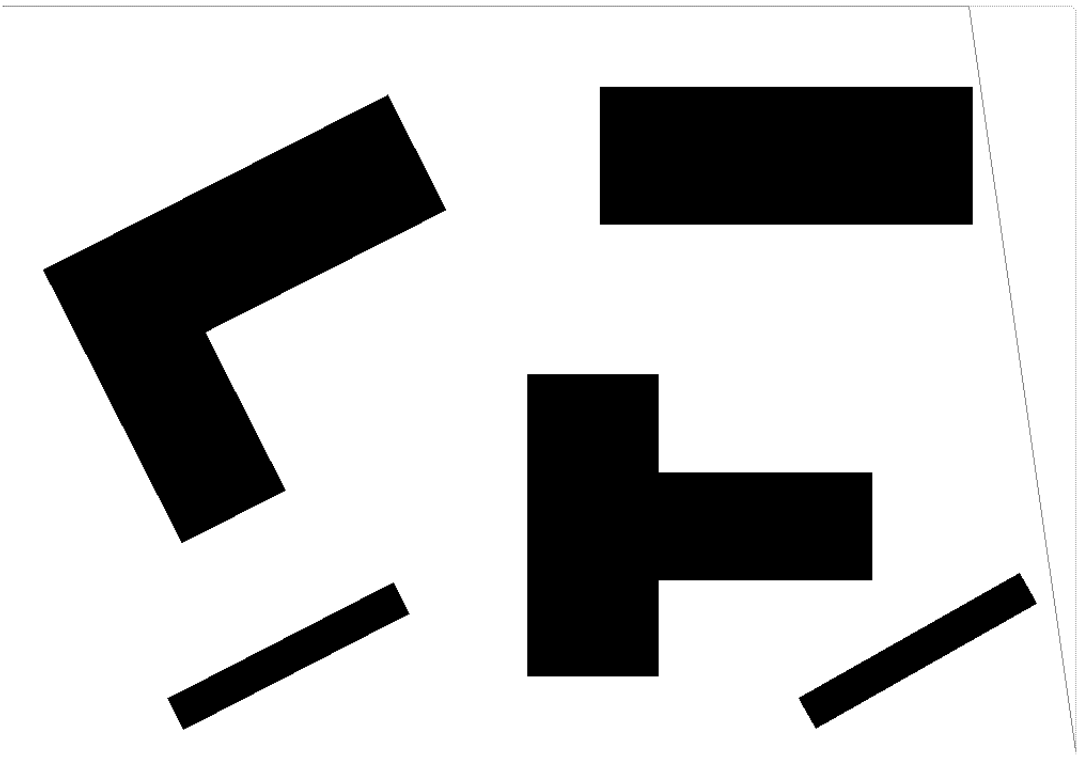

Figura 8: Caminho da melhor escolha com Manhattan para o T.1.

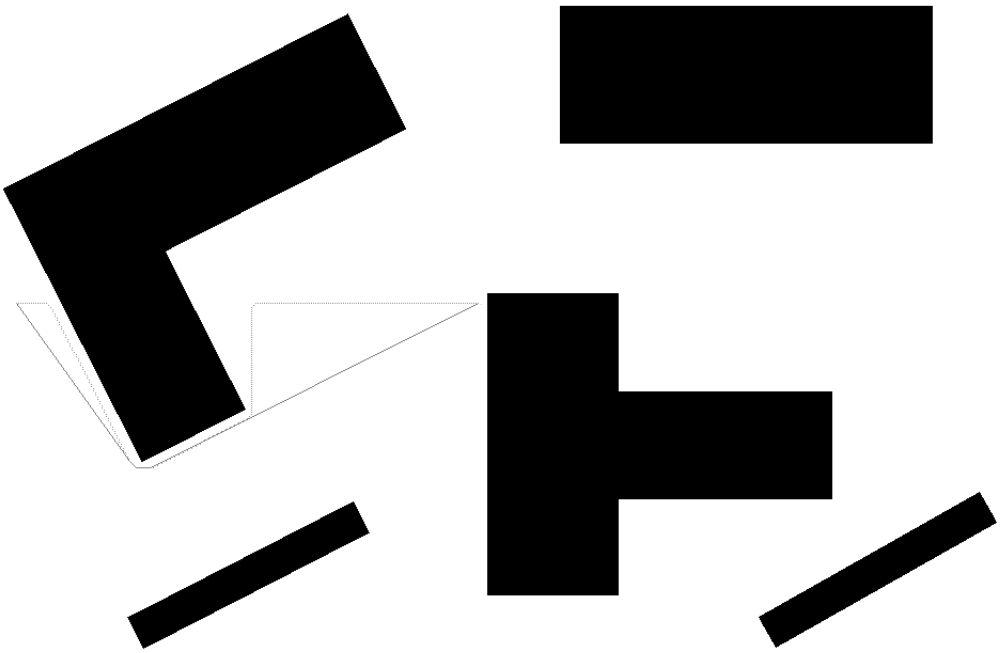

Figura 9: Caminho da melhor escolha com Manhattan para o T.2.

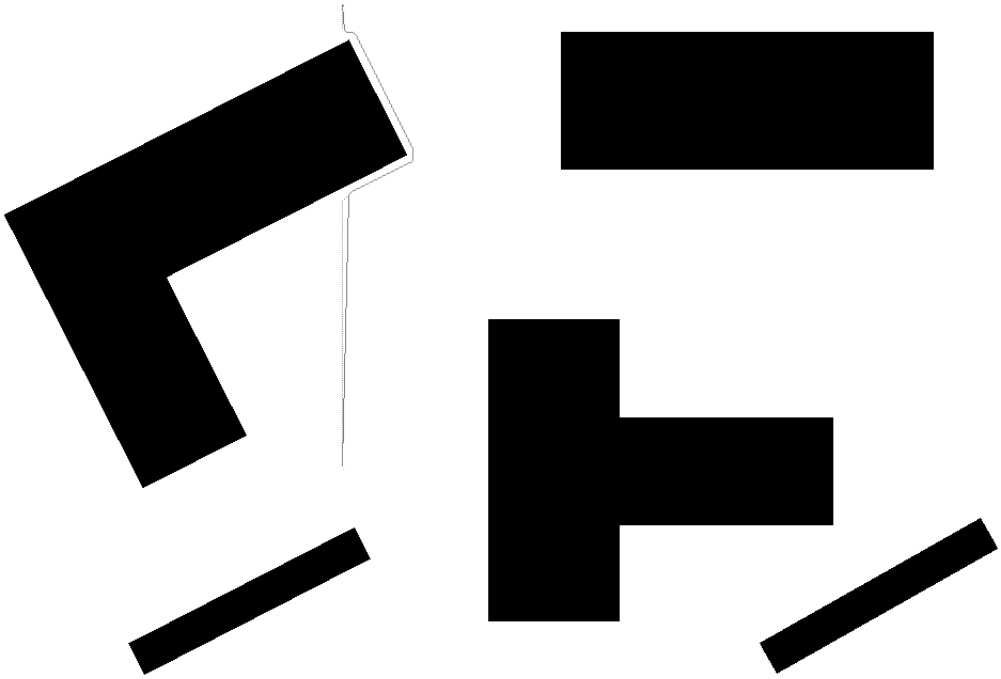

Figura 10: Caminho da melhor escolha com Manhattan para o T.3.

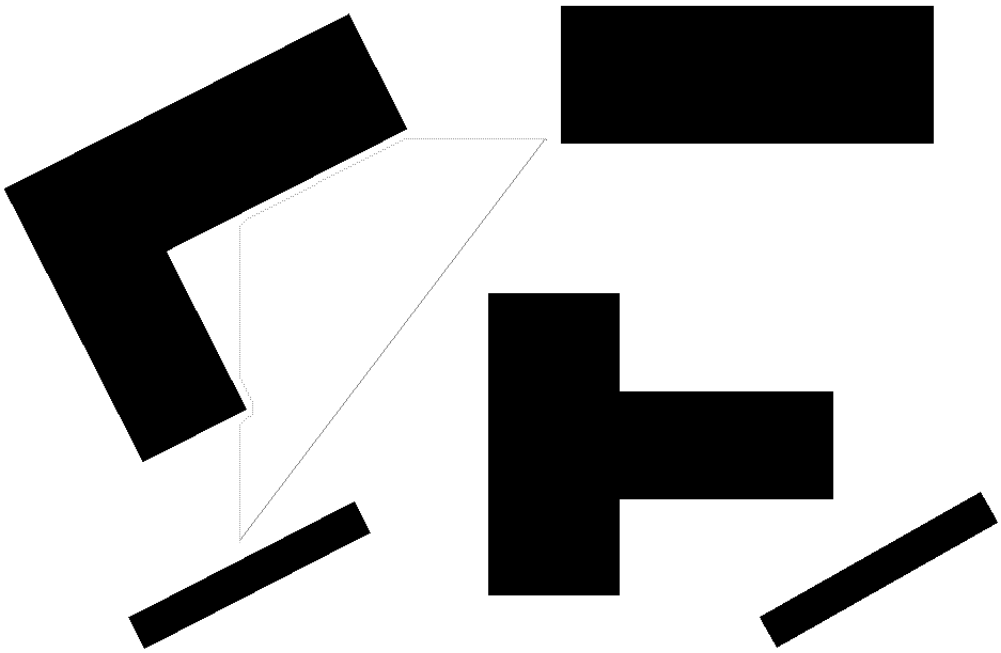

Figura 11: Caminho da melhor escolha com Manhattan para o T.4.

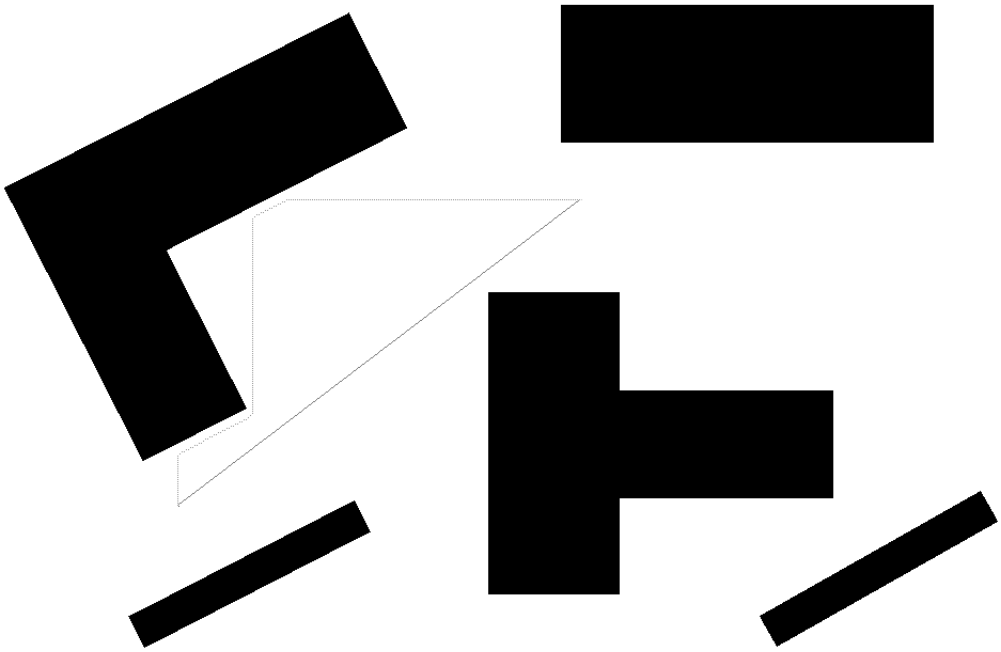

Figura 12: Caminho da melhor escolha com Manhattan para o T.5.

Para a distância euclideana, o tempo de execução foi muito maior que qualquer outro algoritmo. Porém, foi a que gerou os melhores caminhos. Apesar disso, após linearização, os caminhos ficaram praticamente iguais, mostrando que linearização pode melhorar muito uma solução subótima, levando mais a pena buscar uma solução subótima mas rápida e depois pós-processar com linearização.

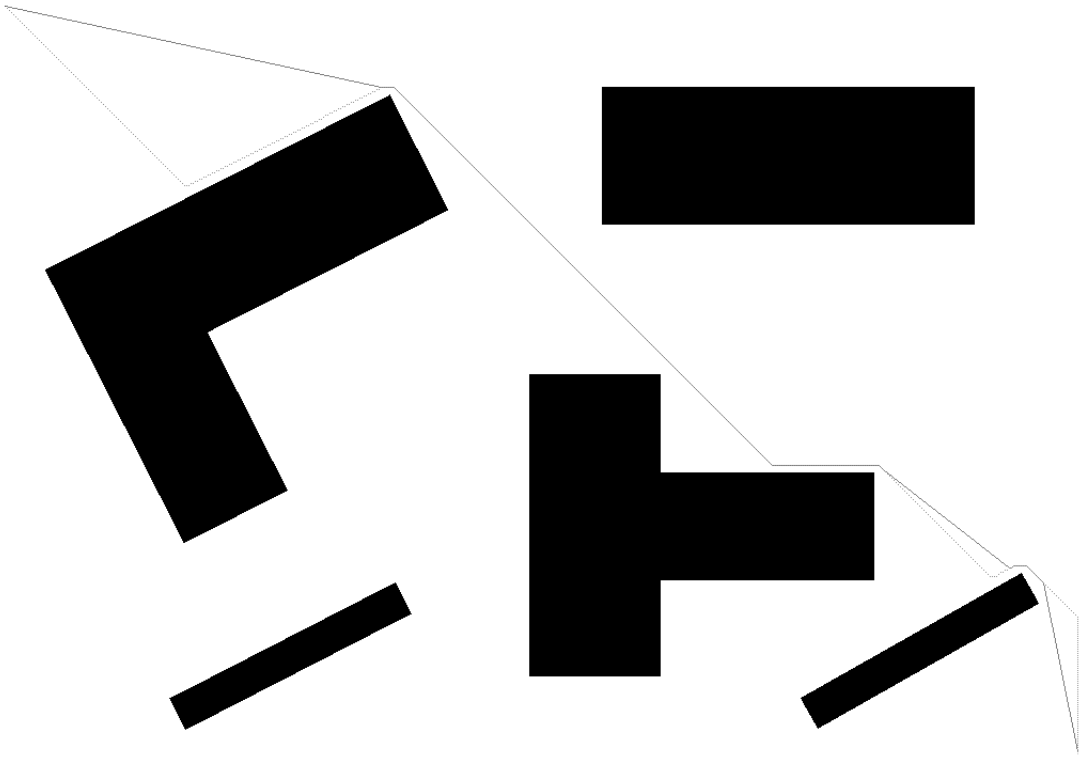

Figura 13: Caminho da melhor escolha com Euclideana para o T.1.

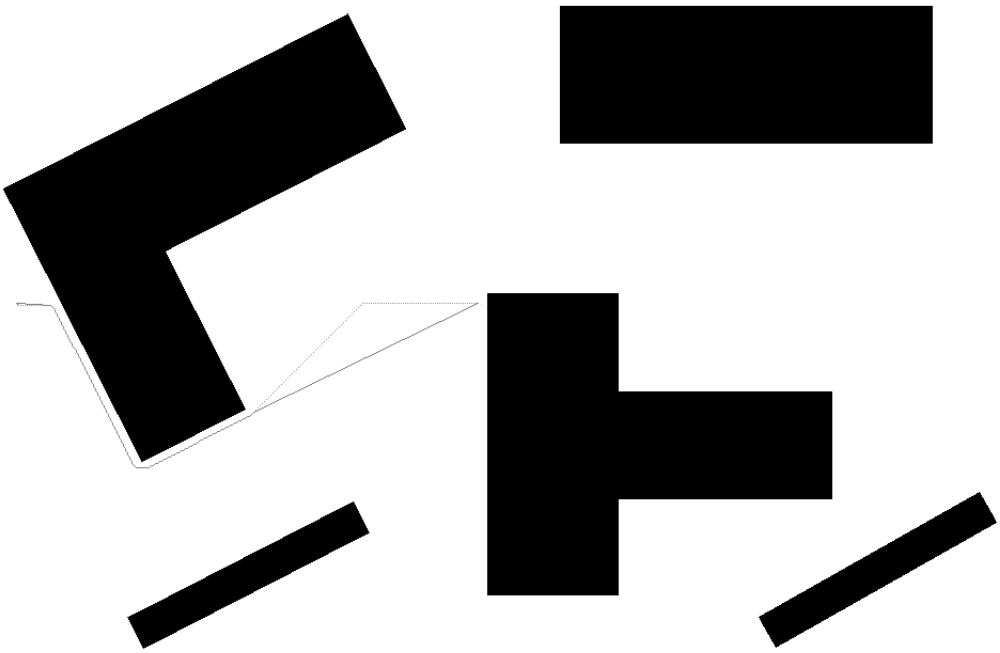

Figura 14: Caminho da melhor escolha com Euclideana para o T.2.

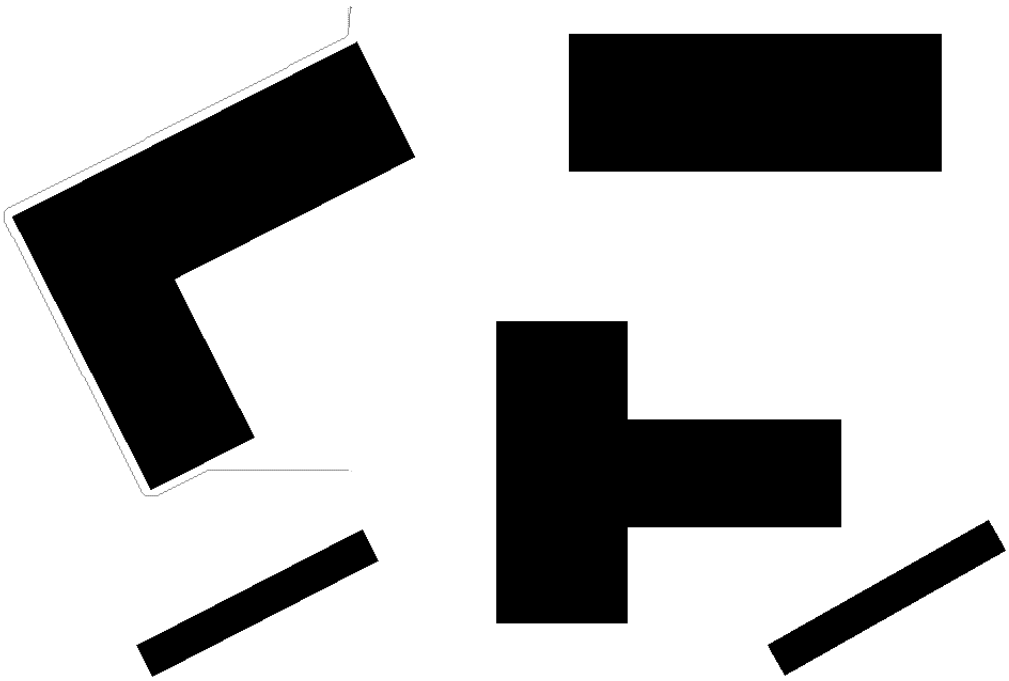

Figura 15: Caminho da melhor escolha com Euclideana para o T.3.

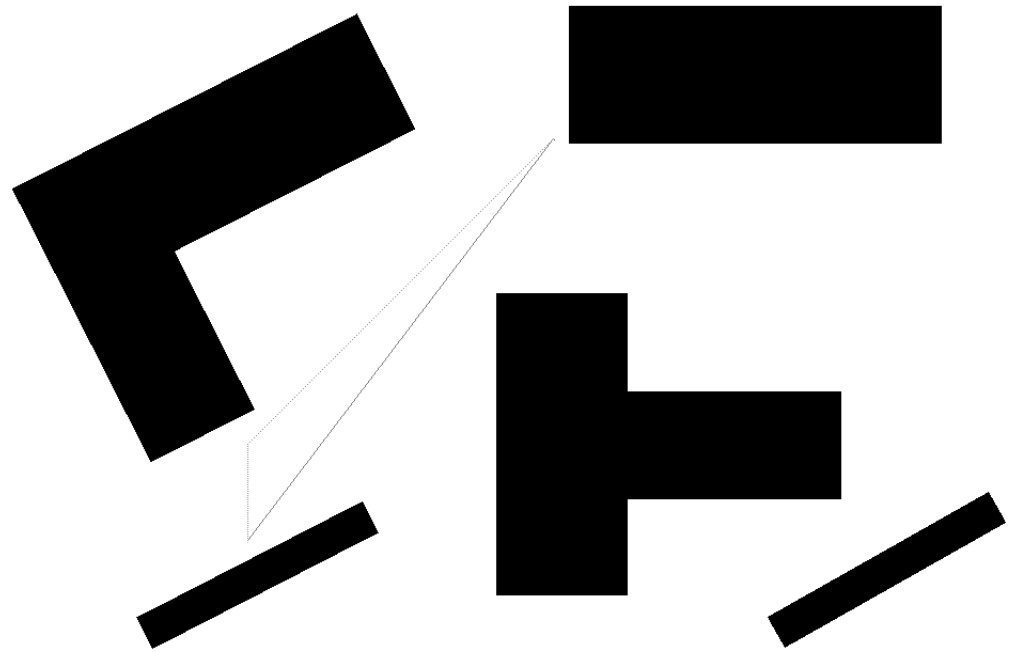

Figura 16: Caminho da melhor escolha com Euclideana para o T.4.

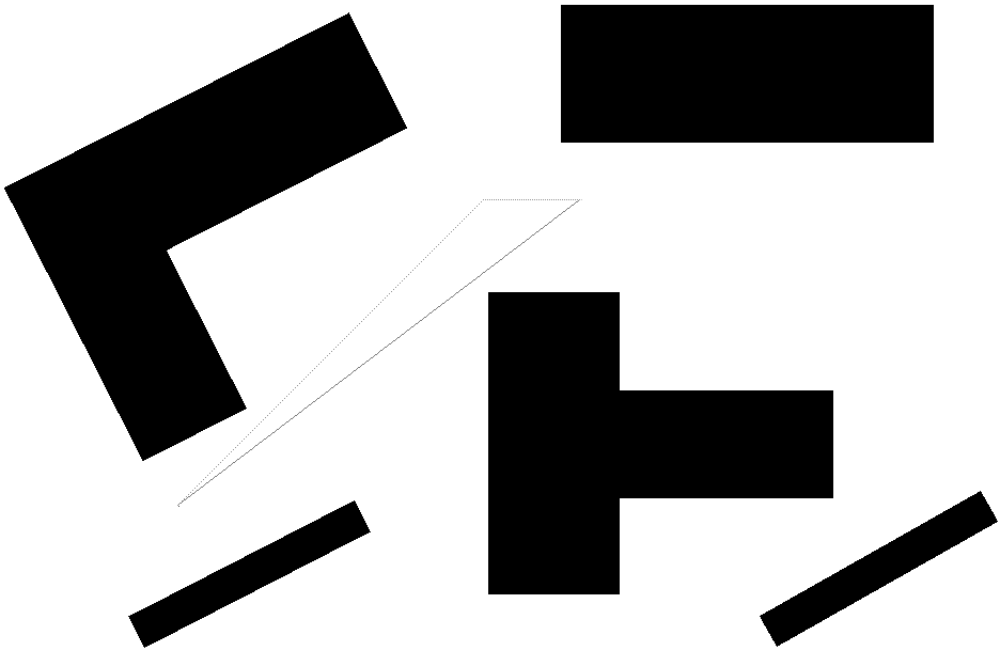

Figura 17: Caminho da melhor escolha com Euclideana para o T.5.

6 Campo potencial

O algoritmo de campo potencial foi implementado pela função `potential`, junto com seu equivalente para visualização `show_potential_field`. O algoritmo aceita como parâmetros: k_a, a constante de atração; k_r, a constante de repulsão; α, a taxa de atualização; ρ, a distância máxima de influência; ϵ, o critério de convergência; p, o ponto inicial e q o ponto final. Para os testes, foram usados os seguintes valores para os parâmetros:

$k_a = 0.1$

$k_r = 1000$

$\alpha = 0.1$

$\rho = 200$

$\varepsilon = 1.0$

Em questão de tempo de execução, o campo potencial foi o mais rápido na maior parte dos casos. O algoritmo de campo potencial foi o único a não conseguir completar todos os

testes, já que não completou o T.1. Foi implementado um jeito de tentar sair do ótimo local, mas mesmo assim, em alguns pontos o algoritmo ainda ficava preso, então foi decidido não deixar no código final. Mesmo assim, deixei o código no Apêndice.

As trajetórias do campo potencial, como era de se esperar, foram as mais "suaves". Mesmo assim, após linearização ficaram parecidas com os outros algoritmos. Em alguns casos, a linearização teve problema com os pontos, e acabaram causando linearizações erradas. Como foi implementada a versão usando a força, não seria necessária a linearização.

Como o campo potencial não consegue completar o T.1, foi omitido o teste e mostrado os restantes.

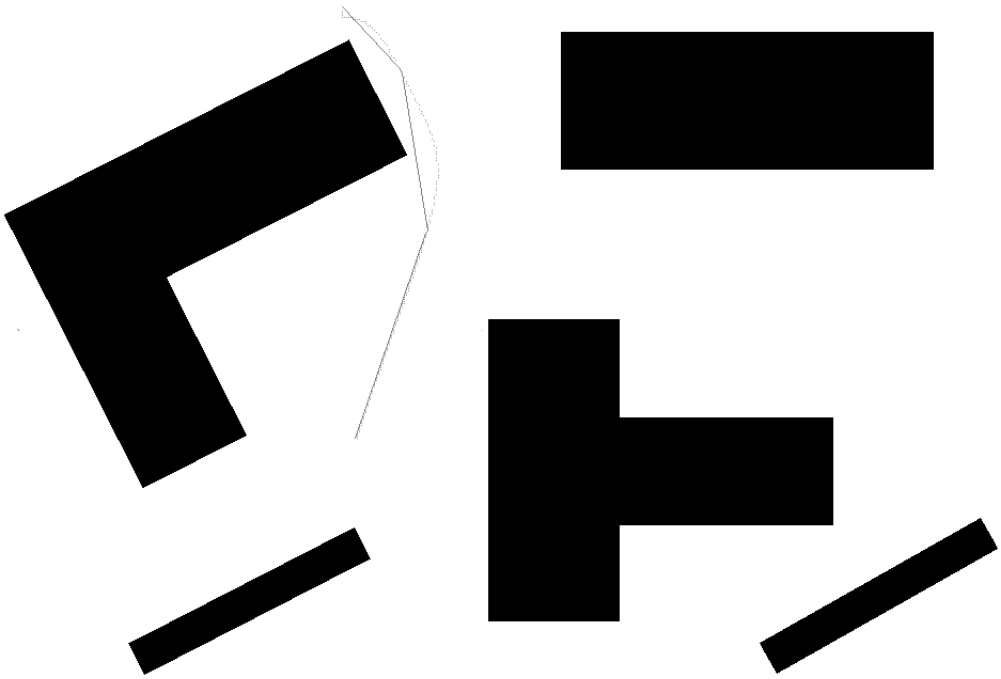

Figura 18: Caminho do campo potencial para o T.2.

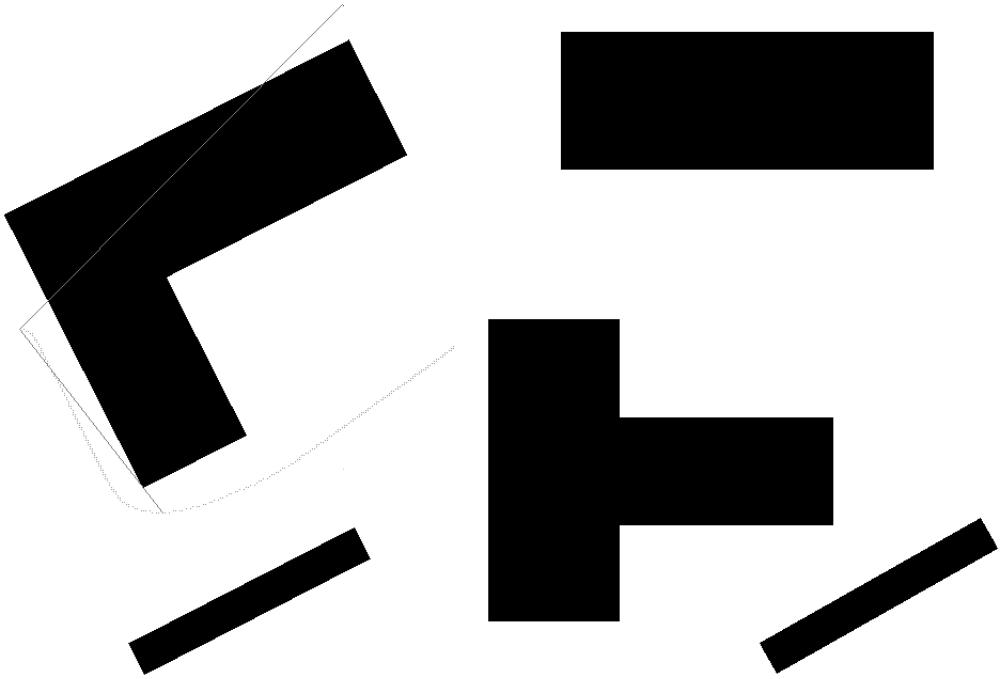

Figura 19: Caminho do campo potencial para o T.3.

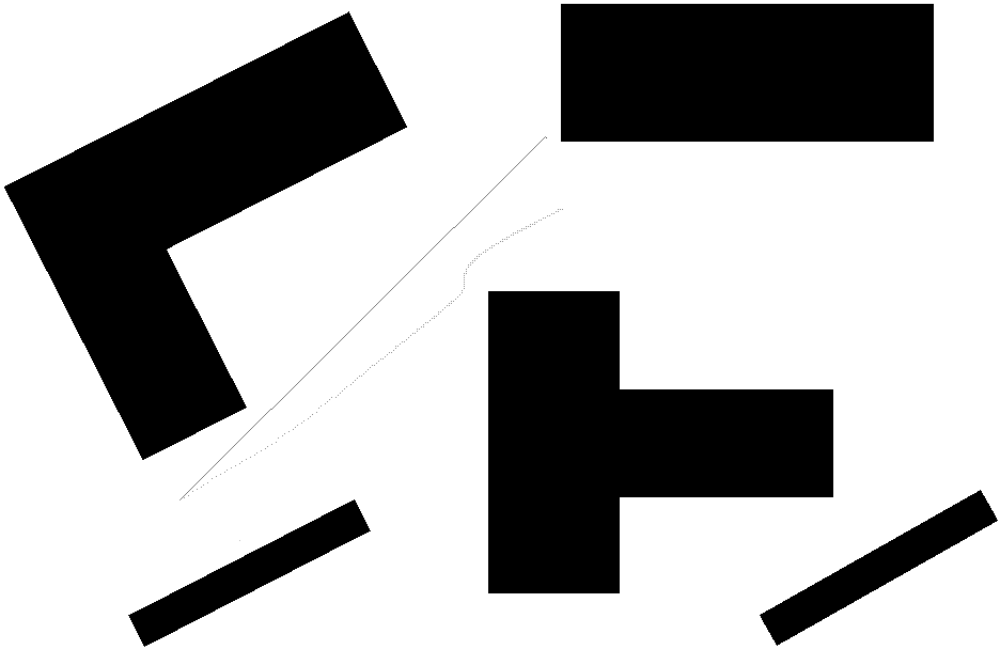

Figura 20: Caminho do campo potencial para o T.4.

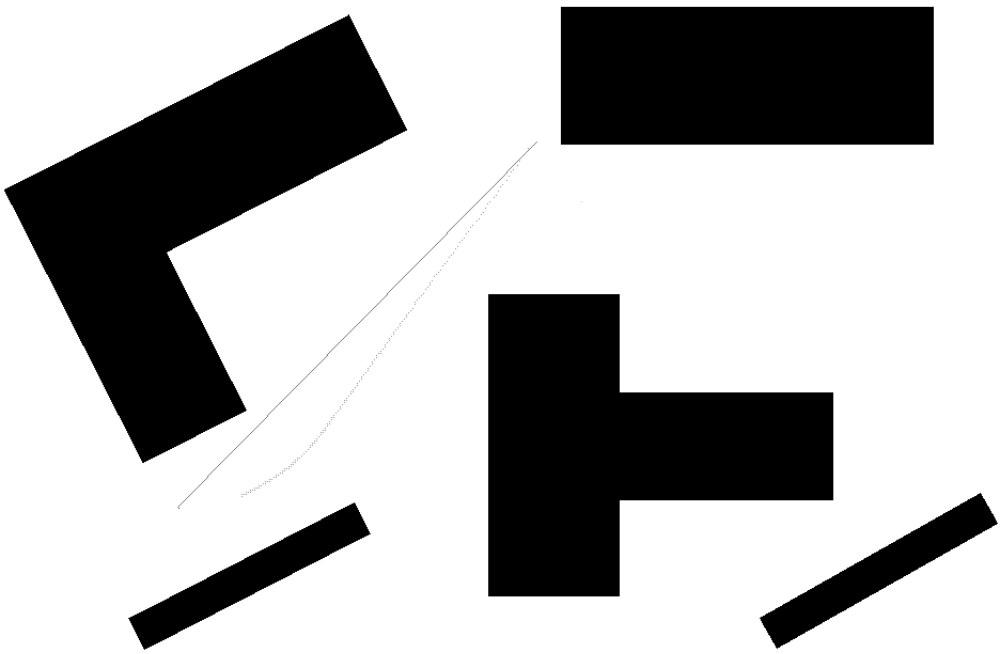

Figura 21: Caminho do campo potencial para o T.5.

7 Apêndice

Código para tentativa de sair de um ótimo local. A ideia era tomar os vetores ortogonais (no caso normalizamos, então temos os ortonormais) ao vetor força. Existem dois, um que segue uma direção parecida à força de repulsão e outra que está na direção contrária da repulsão. Tome por u e v os vetores ortonormais ao vetor força f, e chame de f_r a força de repulsão. Então queremos o vetor força

$$\arg\min_{p\in\{u,v\}}\{||u - f_r||, ||v - f_r||\},$$

já que não queremos ir em direção a obstrução. Após tomarmos p deste mínimo, aplicamos uma constante nele e somamos à posição atual. O problema desta solução é achar uma constante grande o suficiente para tirar o algoritmo do ótimo local e ao mesmo tempo não estourar as coordenadas.

```
def potential(self, k_att, k_rep, alpha, rho, eps, sx, sy, tx, ty):
  C = []
  P = [(sx, sy)]
  for c in self.L:
    C.append((np.array([c[0], c[1]]), np.array([c[2], c[3]])))
  p = np.array([sx, sy])
  g = np.array([tx, ty])
  f = np.inf
  last = p
  n = linalg.norm(f)
  d = linalg.norm(p-g)
  while n > eps or d > 50.0:
    att, rep = self.f_att(p, g, k_att), self.f_rep(k_rep, rho, p, C)
    f = att+rep
    p = p+alpha*f
    if not np.array_equal(last.astype(int), p.astype(int)) and \
       (0 <= p[0] <= self.w and 0 <= p[1] <= self.h):
      print(p, self.orig_pos(p))
      P.append((p[0], p[1]))
      last = np.copy(p)
    n = linalg.norm(f)
    d = linalg.norm(p-g)
    if n <= 2 and d > 50.0:
      # Give it a push. Find orthogonal vector and push in this direction.
      r = np.random.randn(2)
      r -= r.dot(f)*f / np.linalg.norm(r)**2
      r /= np.linalg.norm(r)
      dr, ds = np.linalg.norm(rep-r), np.linalg.norm(rep+r)
      df = -r if dr < ds else r
      p = p+20*df+alpha*f
      n = linalg.norm(f)
  P.append(g)
  P = list(np.unique(P, axis=0).astype(int))
  return P
```