Departamento: PEF – Engenharia de Estruturas e Fundações
Bolsista: Daniel Jorge Caetano
Orientador: Edgard Sant'Anna de Almeida Neto
Projeto: Estudo da Disposição dos Pilares no Subsolo de Edifícios Visando a Maximização das Vagas de Estacionamento

RELATÓRIO FINAL

Estudo da Disposição dos Pilares no Subsolo de Edifícios Visando a Maximização das Vagas de Estacionamento

Bolsista: Daniel Jorge Caetano
Orientador: Edgard Sant'Anna de Almeida Neto

Índice

1. Introdução **3**
2. Pesquisa bibliográfica **4**
3. Entrevista **4**
4. Características relevantes **5**
4.1. Vagas 5
4.2. Vias 6
4.3. Obstáculos 6
4.4. Disposição 6
4.5. Avaliação do estacionamento 6
5. Formulação do problema **7**
5.1. Hipóteses simplificadoras 7
5.2. Discretização do problema 8
5.3. Exemplo 8
6. Classes desenvolvidas **10**
6.1. A classe garage 10
6.2. A classe veicle 11
6.3. A classe obstacle 11
6.4. A classe access 11
6.5. A classe basepath e a classe path 11
6.6. A classe basestall e a classe stall 11
7. Cronograma **12**
8. Observações finais **12**
9. Bibliografia **13**
Anexo A. Definição do quadriculamento **14**
Anexo B. Estudos adicionais **17**
B.1. Ordem de disposição geral 17
B.2. Disposição das vias 17
B.3. Disposição das vagas 18
Anexo C. Headers das classes principais **23**
Anexo D. Instruções de uso do software desenvolvido **30**
D.1. Apresentação 30
D.2. Operação 30
D.3. Arquivos de dados 32
D.4. Exemplos 36

1. Introdução

Até meados do século XX, o arranjo estrutural das edificações residenciais, definido pela planta baixa do andar tipo, era semelhante ao arranjo adotado para a infra-estrutura. Com o crescimento do número de automóveis por família nas últimas quatro décadas, a demanda por vagas de estacionamento aumentou. Este aumento fez com que o arranjo estrutural da infra-estrutura passasse a depender cada vez mais da disposição das vagas de estacionamento. Como o arranjo estrutural da superestrutura continuou sendo definido pela planta do andar tipo, criou-se a necessidade de dispendiosas estruturas de transição para transmitir os esforços da superestrutura para a infra-estrutura.

Este estudo visou a determinação de uma possível sistematização para o processo de disposição de vagas no subsolo de um edifício residencial, através da criação de um programa que realize este processo automaticamente, tornando viável o estudo de diversas soluções a um baixo custo. A comparação entre essas soluções pode proporcionar a busca de uma solução que maximize o número de vagas e minimize o custo da estrutura de transição do edifício. Uma outra vantagem advinda da existência de diversas soluções é a possibilidade de arquitetos e projetistas de estruturas as utilizarem como instrumento para a determinação das regiões onde a disposição de pilares interferirá minimamente na distribuição ótima de vagas.

O programa foi desenvolvido em uma linguagem orientada a objetos (C++), utilizando-se de classes elaboradas para a descrição dos elementos relevantes na resolução do problema. Tal programa é capaz de resolver problemas que respeitem as hipóteses adotadas neste trabalho, validando a adequação destas classes ao cumprimento dos objetivos propostos.

Neste relatório são apresentadas a pesquisa bibliográfica realizada, uma entrevista realizada com um arquiteto especialista na disposição de vagas em estacionamento, as características relevantes levantadas para a descrição e solução do problema, bem como a formulação do mesmo e sua solução. Finalmente é apresentado um resumo das classes desenvolvidas, cronograma e comentários finais.

2. Pesquisa Bibliográfica

Uma pesquisa bibliográfica abrangente foi realizada nas bibliotecas da USP (EPUSP, FAU, EPEC, EESC) em busca de livros e trabalhos publicados sobre o tema deste estudo ou associados a ele. Foram encontrados poucos trabalhos que de alguma forma se relacionassem com a disposição de vagas em subsolos de edifícios. Por meio dos resumos dos trabalhos, percebeu-se que versavam sobre estimativas de vagas em grandes estacionamentos de superfície, estudos sobre estacionamento de veículos auxiliados por computador etc.

Entretanto, alguns livros específicos sobre o assunto foram encontrados (ver bibliografia) e foram devidamente analisados com o objetivo de levantar as principais características para a formulação do problema e eventuais soluções. Embora fossem livros antigos, várias informações relevantes foram obtidas e se encontram resumidas na Seção 4.

3. Entrevista

Com o objetivo de complementar a pesquisa bibliográfica e também verificar as tendências atuais na área de disposição de vagas em subsolo de edifícios, foi entrevistado um arquiteto especialista em distribuição de vagas de estacionamento. A fim de melhorar o aproveitamento da entrevista, foram realizados diversos estudos manuais de disposição de vagas em áreas com diferentes graus de complexidade geométrica (Anexo B). Com estes estudos, foi possível perceber quais eram as principais dificuldades no processo, bem como levantar as interferências causadas pelos elementos estruturais na disposição de vagas. Tais estudos originaram questões que foram respondidas durante a entrevista.

A entrevista, realizada a 3 de Dezembro de 2001, às 8:30 da manha, no escritório da ***Michel Sola Consultoria e Engenharia S/C Ltda***. (Rua Bandeira Paulista, 716, 11^{o} andar) durou cerca de uma hora. Durante este período foi possível conversar com o arquiteto e outros funcionários da área técnica da empresa. Como resultado das perguntas e discussão sobre as plantas previamente estudadas, várias informações relevantes puderam ser obtidas, principalmente sobre o traçado das vias, avaliação do estacionamento, etc. (veja Seção 4). Sérgio M. Sola também apresentou plantas com soluções geradas por sua empresa, comentando os problemas e suas soluções.

4. Características Relevantes

Com a pesquisa bibliográfica, os estudos manuais e as informações obtidas na entrevista, foram definidas as características relevantes na descrição do problema, tendo ficado claro que a disposição geral é influenciada principalmente pelas características geométricas de vias, vagas e obstáculos [1]. A hierarquia dos elementos pode ser verificada na figura 4.1.

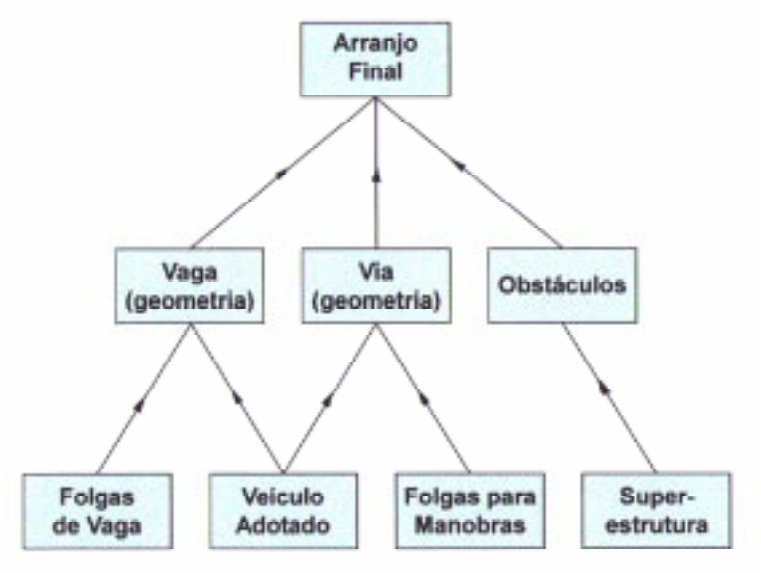


Figura 4.1 – Relações entre os elementos

4.1. Vagas

A largura e o comprimento da vaga dependem do tamanho do veículo de projeto, que é o maior veículo que fará uso freqüente do estacionamento [1]. Deve ser adotado um tamanho único de vaga e não deve ser utilizado um *valor médio*.

A largura da vaga deve levar em consideração folgas para retrovisores [1], abertura de portas [1,3], entrada e saída de passageiros [1], não deixando de considerar a necessidade de embarque e desembarque de carrinhos de bebês, cadeiras de roda etc. No caso da abertura de portas, pode ser considerado o espaço adicional da folga da vaga vizinha se ela existir. Caso a via de acesso seja estreita, é necessário alargar as vagas para permitir as manobras necessárias [1]. É sugerido adotar uma folga extra de 10 cm [1].

O comprimento da vaga deve ser considerado igual ao do veículo de projeto, sendo permitido que veículos maiores ocupem uma pequena parcela da via de acesso [1].

4.2. Vias

A medida fundamental da via é a largura, que depende do veículo de projeto, mas deve considerar também outros fatores, como os espaços adicionais necessários às manobras dos veículos [1]. A largura das vias deve ser aumentada quando forem adotadas vagas pequenas [1] ou as velocidades permitidas nas vias forem altas [3], de forma que a via forneça espaço para a realização das manobras com segurança. Vias mais estreitas são mais facilmente alocadas e interferem menos com os elementos estruturais.

A alocação das vias deve ser feita de modo que o traçado resultante tenha o menor comprimento e o menor número de curvas [5]. O traçado básico deve ser definido paralelamente à maior direção do contorno do subsolo, com vias transversais posicionadas de modo a minimizar as distâncias necessárias para os veículos alcançarem as vagas [2]. Definida a direção de uma via, seu posicionamento transversal numa área é ditado pelos obstáculos menores.

4.3. Obstáculos

Deve-se reduzir ao máximo o número de obstáculos, procurando posicioná-los de modo a evitar a região central do estacionamento, bem como posições *incômodas* como as regiões de aberturas de portas etc. [2]. A disposição dos outros elementos deve respeitar a disposição dos obstáculos.

Os obstáculos podem ser classificados em grandes e pequenos. Os obstáculos grandes têm pelo menos uma das dimensões de mesma ordem das dimensões da geometria da garagem (paredes de contorno, grandes pilares, caixa de elevador, etc.). Os pequenos obstáculos são aqueles que não atendem a esta condição (pequenos pilares e outras restrições menores).

4.4. Disposição

A disposição deve ser iniciada pela alocação das vias, sendo as vagas alocadas posteriormente [5]. Em grandes estacionamentos deve ser prevista uma área de acumulação na entrada [5], além de ser facilitada a recirculação de veículos que não encontrem vagas disponíveis [5]. Um melhor aproveitamento é obtido com a disposição das vagas a 90^o na região próxima ao contorno [2].

4.5. Avaliação do Estacionamento

O critério fundamental para determinar a qualidade da solução do problema é o número total de vagas. Entretanto, ao se maximizar o número de vagas, deve ser contemplada a facilidade das manobras de estacionamento [5].

As vias devem ser curtas, minimizando o trajeto de acesso às vagas. Sempre que possível, elas devem ser retilíneas, pois um grande número de curvas no trajeto de estacionamento diminui a qualidade do mesmo [5].

O índice de aproveitamento de espaço (área ocupada por vagas em relação à área ocupada por vias) deve ser máximo [5]. A análise da solução com e sem obstáculos pequenos pode ser bastante positiva se o objetivo for avaliar o impacto da disposição dos elementos estruturais no número total de vagas.

Nota: *1, 2, 3 e 4 referem-se às publicações na bibliografia. 5 refere-se às observações do arquiteto Sérgio M. Sola.*

5. Formulação do Problema

Com base nos tópicos da Seção 4 e estudos adicionais visando aperfeiçoar as regras de disposição (Anexo B), foi possível formular o problema da disposição de vagas no subsolo de edifícios. Devido à complexidade do problema, ele foi decomposto em elementos mais simples (garagem, vias, vagas etc.) e foram adotadas algumas hipóteses simplificadoras que permitiram a elaboração de classes em linguagem orientada a objetos. Essas classes descrevem os aspectos relevantes do problema e seus elementos. Um resumo destas classes pode ser encontrado na Seção 6.

Com base nas hipóteses simplificadoras foi definido um problema exemplo a ser resolvido. Para resolvê-lo, foi desenvolvido um programa baseado nas classes elaboradas.

5.1. Hipóteses Simplificadoras

Para tornar possível a elaboração e a implementação das classes, foram adotadas as seguintes hipóteses simplificadoras:

- os espaços livres sob as rampas de acesso são desprezados;
- os obstáculos grandes próximos ao contorno são anexados ao mesmo;
- os obstáculos grandes internos são descritos como simples obstáculos;
- os contornos externos são admitidos paralelos aos eixos coordenados;
- os estacionamentos têm as áreas usuais de edifícios residenciais (menores que 1000 m^2);
- cada sub-região retangular do estacionamento possui uma única via;
- os veículos podem estacionar de ré;
- as vias têm largura suficiente para os veículos realizarem as manobras de estacionamento.

5.2. Discretização do Problema

Com os estudos realizados (Anexo B), concluiu-se que a discretização do espaço do problema poderia ser bastante positiva para sua solução. Por simplicidade de implementação, foi adotada a discretização na forma de quadriculado.

Na implementação, foi admitida uma subdivisão de 225 cm de modo que as vias tivessem uma largura de duas unidades e as vagas ocupassem dois quadrados contíguos. Os detalhes da obtenção destes valores encontram-se no Anexo A.

5.3. Exemplo

Com base nas hipóteses simplificadoras, foi adotada uma geometria de garagem representativa (Figura 5.3.1), sobre a qual o programa desenvolvido deveria ser capaz de dispor as vias e vagas, respeitando as restrições fornecidas. As soluções geradas pelo programa para essa garagem podem ser observados nas figuras 5.3.2 e 5.3.3.

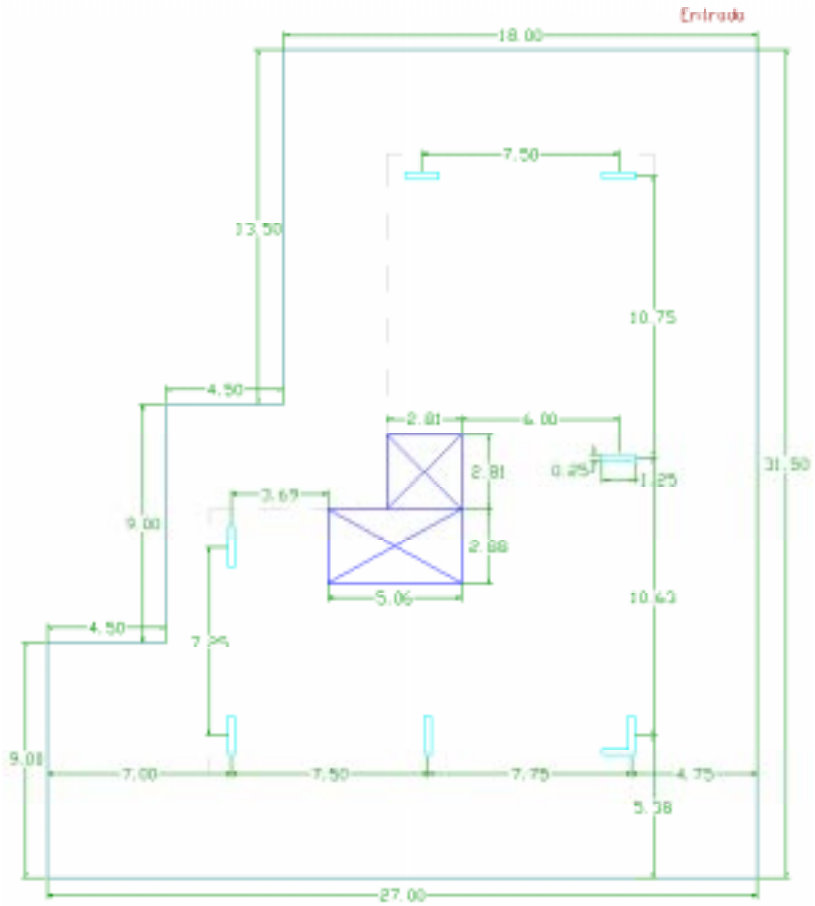


Figura 5.3.1 – Geometria do problema resolvido

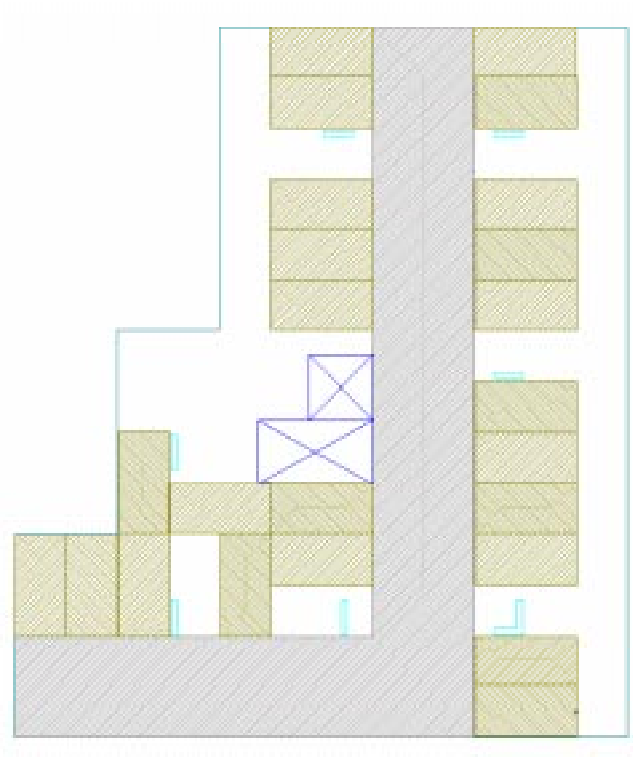


Figura 5.3.2 – Uma possível solução gerada

A solução apresentada na figura 5.3.2 é uma das soluções possíveis para o problema, e foi obtida de forma totalmente automática, respeitando todas as restrições impostas e dentro da discretização adotada.

Refinando-se a malha e priorizando outras diretrizes pode-se obter diferentes soluções para o problema. Entretanto, uma verificação mais imediata de melhoria de aproveitamento dos espaços pode ser feita simplesmente variando a geometria ou obstáculos, buscando-se minimizar grandes áreas desocupadas e procurando mover obstáculos que estejam causando a translação das vias de sua posição ideal.

A figura 5.3.3 ilustra a melhoria que pode ser obtida apenas variando a posição dos obstáculos, onde o edifício foi transladado no terreno, mantendo a posição relativa entre os pilares. Como é possível observar, a mudança provocou um aumento de cerca de 30% no número de vagas (de 24 para 33 vagas), sendo este valor bastante significativo.

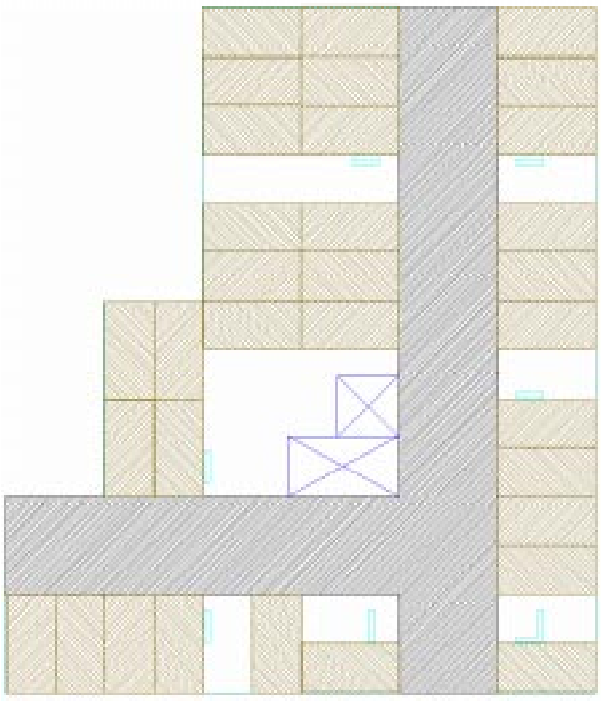


Figura 5.3.3 – Solução com movimentação do edifício

Movendo os pilares relativamente uns aos outros pode-se buscar resultados ainda melhores, sem qualquer tipo de alteração no modelo. Com um conjunto maior de soluções possíveis, pode-se optar por aquela que trouxer maiores benefícios, em termos do número total de vagas e dos custos estruturais envolvidos.

Ambas as soluções foram obtidas de forma automática, tendo sido o resultado numérico convertido manualmente para a forma gráfica em que aqui estão apresentados. Mais detalhes sobre a operação do programa desenvolvido podem ser obtidos no Anexo D.

6. Classes Desenvolvidas

Para a descrição do problema da disposição de vagas em uma área de estacionamento, foram criadas 8 classes principais (**garage**, **veicle**, **obstacle**, **access**, **basepath**, **path**, **basestall** e **stall)**, além de algumas classes geométricas auxiliares (DJ_point, DJ_line, DJ_rectangle e DJ_polyline). A hierarquia de classes pode ser observada na figura 6.1, os headers das mesmas encontram-se no Anexo C. Uma breve descrição de cada uma das classes encontra-se nas subseções a seguir.

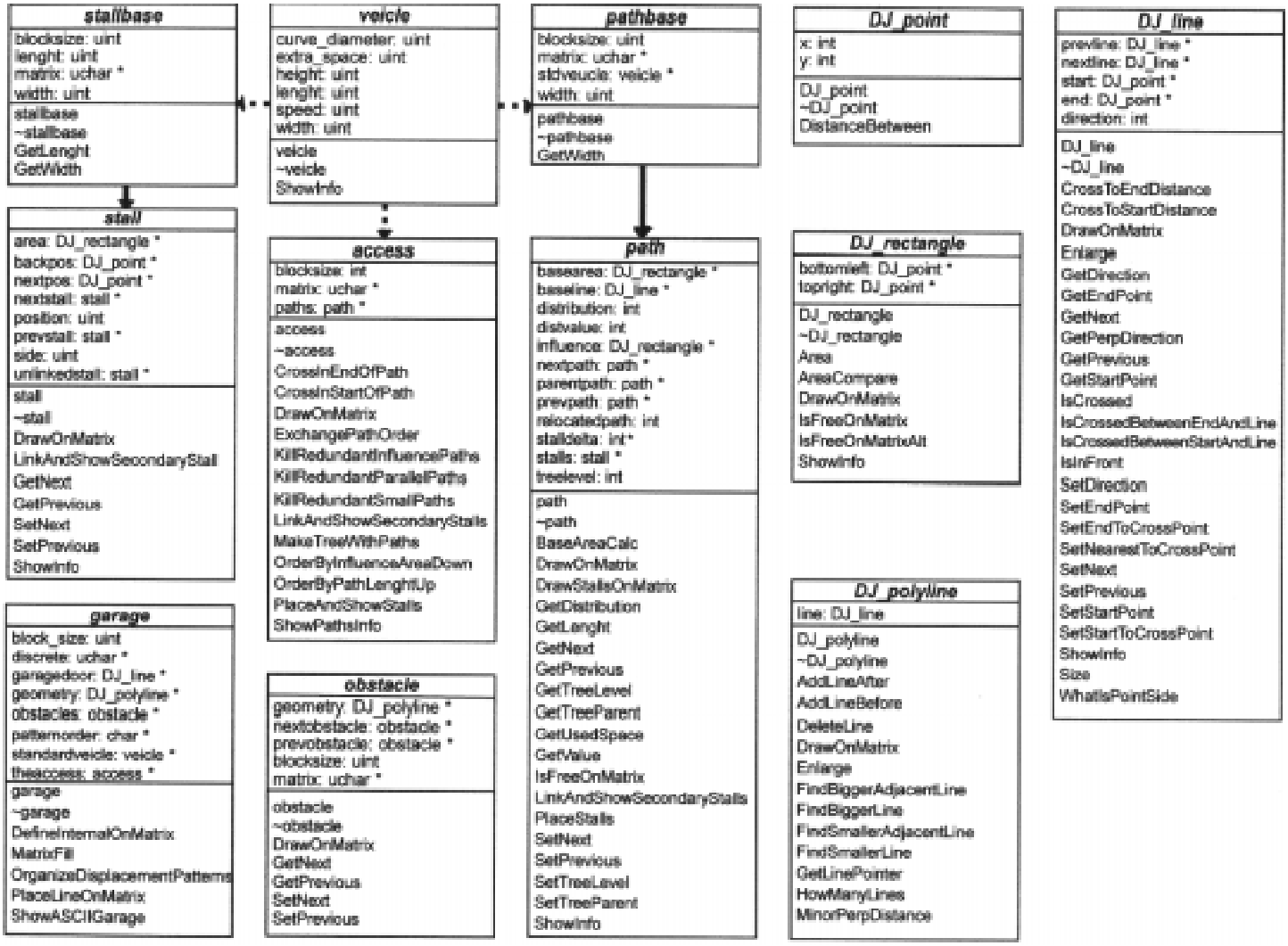


Figura 6.1 – Diagrama UML das classes desenvolvidas

6.1. A Classe Garage

A classe **garage** descreve a geometria da área onde todos os objetos das outras classes atuam, incluindo todas as restrições. O primeiro objeto a ser criado é da classe **garage**, sendo responsável pelo início do processo de disposição. Ele cria os objetos da classe **obstacle** e também o objeto da classe **access**, que dá continuidade ao processo. Este é também o objeto que fornece os resultados finais da disposição.

6.2. A Classe Veicle

A classe **veicle** descreve as dimensões do veículo de projeto, incluindo largura, comprimento, folgas, raio de curvatura etc.

6.3. A Classe Obstacle

A classe **obstacle** descreve obstruções espaciais estáticas, que devem ser evitadas por objetos de outras classes.

6.4. A Classe Access

A classe **access** descreve toda a via de circulação desde a entrada até a última vaga do estacionamento. O objeto desta classe tem a função de disparar os objetos da classe **path** (vias) que o compõem. Após alocar as vias, o objeto da classe **access** comanda os objetos **path** a dispararem os objetos da classe **stall** (vagas).

6.5. A Classe BasePath e a Classe Path

A classe **basepath** descreve os atributos de uma via que independem de posição, como a largura. A classe **path** é derivada da classe **basepath** e descreve uma via isolada completa, incluindo sua posição, comprimento, área de influência etc. Os objetos da classe **path** interagem diretamente com os obstáculos, desviando-se dos mesmos em busca da melhor posição possível. Os objetos da classe **path** disparam objetos da classe **stall**, admitidos como pertencentes à via nesta implementação.

6.6. A Classe BaseStall e a Classe Stall

A classe **basestall** descreve os atributos de uma vaga que independem de posição, como a largura e o comprimento. A classe **stall** é derivada da classe **basestall** e adiciona àquela os elementos dependentes de posição de uma vaga. O objeto da classe **stall** também interage com os obstáculos, não permitindo sua criação sobre uma obstrução. Objetos **stall** com atributo perpendicular à via também são responsáveis pela criação de um outro objeto **stall** subseqüente a ele, em espaços disponíveis da área de estacionamento.

7. Cronograma

Na etapa inicial da pesquisa, foi proposto um cronograma que incluía períodos para a pesquisa bibliográfica, estudo do modelo, definição das classes e implementação destas em um programa.

As etapas iniciais (pesquisa bibliográfica e estudo do modelo) foram concluídas no prazo estabelecido e a elaboração das classes foi iniciada. Entretanto, o material inicialmente pesquisado não atendeu completamente às expectativas e maiores estudos foram necessários.

Apesar do pequeno atraso no início da elaboração das classes, esta etapa foi concluída com sucesso. As classes e atributos foram sendo complementados durante a elaboração do programa, cujo resultado pode ser verificado nas figuras 5.3.2 e 5.3.3, cumprindo os objetivos desta pesquisa.

8. Observações Finais

Com a pesquisa bibliográfica foi possível definir os principais aspectos relevantes para a resolução do problema. Entretanto, para que ele fosse descrito de forma adequada à sua solução, alguns estudos adicionais tiveram de ser realizados.

Com a adoção de hipóteses simplificadoras, foi possível delimitar o problema a ser resolvido e, ao mesmo tempo, viabilizar a elaboração das classes e a implementação do programa. Os resultados foram bastante satisfatórios e comprovam que o assunto pesquisado é bastante promissor. Eles também evidenciaram a importância dos elementos descritos pelas classes na resolução do problema.

O programa desenvolvido pode ser estendido para resolver qualquer tipo de geometria (áreas maiores que de edifícios residenciais com paredes não necessariamente perpendiculares), além da aplicação de algoritmos de otimização local e global de forma a não só aumentar a abrangência do mesmo, mas também melhorar a qualidade de suas soluções. Uma outra linha de estudo abordaria a avaliação automática do estacionamento gerado, visando a escolha de disposição de vagas para estacionamentos de mesma geometria mas com diferentes arranjos de pilares.

Acredita-se que os resultados desta pesquisa podem ser aprimorados e utilizados para possibilitar uma melhor integração entre os projetos de andar tipo, superestrutura e estacionamento, visando a melhoria em sua qualidade e a redução substancial dos custos finais da estrutura como um todo.

9. Bibliografia

1. Ricker, Edmund R. *Traffic Design of Parking Garages.*
   Saugatuck, Conn., Eno Foundation for Highway Traffic Control, 1957.

2. National Parking Assoc. Parking Consultants Council. *The Dimensions of parking.*
   Washington: ULI-the Urban Land Institute, 1979.

3. Baker, Geoffrey Harold. *Parking.*
   Imprenta New York, Reinhold, 1958.

4. *Código de Obras e Edificações da Cidade de São Paulo.*
   Imprensa Oficial, http://sampa3.prodam.sp.gov.br/sehab/codigo/codigo.asp.

Anexo A

Definição do Quadriculamento

Após a realização de estudos manuais de distribuição de vagas de estacionamento, ficou claro que a adoção de uma discretização dos elementos simplificaria muito a descrição e resolução computacional do problema.

Por simplicidade de implementação, adotou-se o quadriculamento como discretização da área disponível para a disposição de vagas. Com o objetivo de verificar quais seriam as dimensões ideais de trabalho para os elementos desta malha, foram realizados alguns estudos, os quais estão resumidos a seguir.

Os valores mais importantes a serem representados pelos elementos são as dimensões de vagas e vias, as quais ocupam a maior parte da área do estacionamento. Assim, é importante que os elementos representem da melhor forma possível estes dois entes.

Algumas dimensões básicas para as vias/vagas podem ser encontradas na bibliografia [2,3,4], porém os valores de referência nesta análise foram obtidos em campo:

- *Estacionamento da USP (superfície):* Vagas 2,35x5,00m; Vias 5,3 m
- *Estacionamento médio (subsolo):* Vagas 2,25x4,60m; Vias 3,5 m

Os veículos verificados possuem as seguintes medidas:

Modelo/Ano	Diâmetro Externo para Curvatura (m)	Comprimento (m)	Largura (c/ Retrovisor) (m)
Astra/00	10,80	4,10	1,99
Ipanema/97	10,50	4,40	1,80
Kadett/97	10,50	4,10	1,81
Parati/84	10,20	4,69	1,62
Tipo/87	10,30	4,00	1,83
Vectra/98	11,30	4,50	1,84
Voyage/84	10,20	4,63	1,60

O veículo adotado como padrão foi o Vectra. A partir das dimensões deste e de seu raio de curvatura foram realizadas diversas experimentações e desenhos, alguns dos quais estão representados nas Figuras A.1 a A.4.

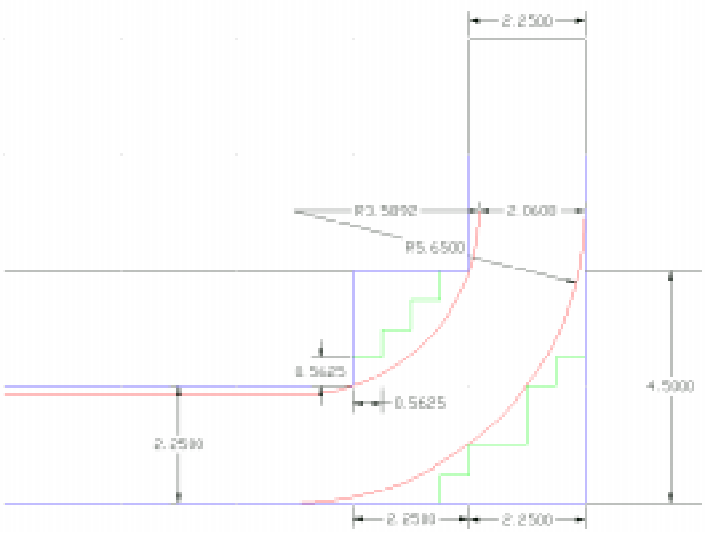


Figura A.1 – Movimento de curva

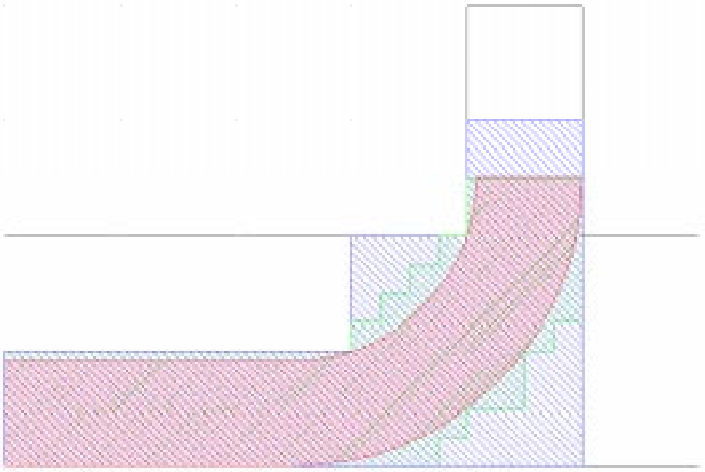


Figura A.2 – Efeito da discretização em movimento de curva

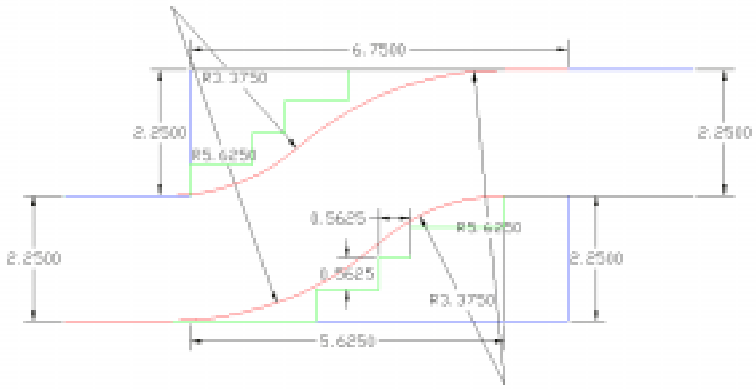


Figura A.3 – Movimento de mudança de faixa

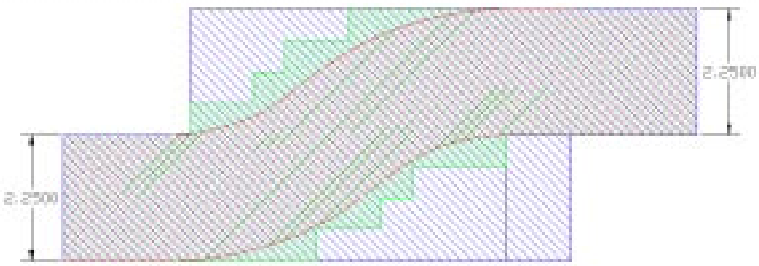


Figura A.4 – Efeito da discretização em movimentos de mudança de faixa

Com esses estudos percebeu-se que o quadriculamento ideal para solução do problema seria a utilização de elementos, com lados de (2,25/4) m, adotando-se vagas de 4x8 elementos (2,25x4,5 m) e vias de 8 elementos (4,5 m) de largura (representada nas figuras na cor verde). Segundo a análise das figura A.1 a A.4, que leva em conta as curvaturas, esta discretização descreve bem o problema, sem perder muita resolução e sem aumentar exageradamente o esforço computacional.

No entanto, esta discretização mostrou-se bastante complexa para uma programação inicial, sendo descartada na presente pesquisa, tendo sido adotada uma discretização menos refinada. A discretização adotada foi a de elementos quadrados de 2,25 m de lado (representada nas figuras utilizando a cor azul), tendo as vagas uma composição de 1x2 elementos, e as vias 2 elementos de largura.

Anexo B

Estudos Adicionais

A fim de formular questões para enriquecer a entrevista realizada (Seção 3) e também com o intuito de compreender melhor as inter-relações entre os elementos do problema da disposição de vagas de estacionamento, foram realizados diversos estudos manuais. O aspecto final de alguns dos estacionamentos desenvolvidos nesta etapa pode ser verificado nas figuras B.1 a B.4.

Durante o preenchimento manual das vagas nestes estacionamentos procurou-se observar, dentre as diversas formas de se iniciar e conduzir uma disposição, quais as melhores e quais as piores maneiras. Além disso, foram levantadas as principais conseqüências e problemas que surgiam em decorrência de cada procedimento executado durante a distribuição, em especial a forma como os obstáculos e geometria influenciam na disposição geral das vagas. Alguns pontos bastante relevantes foram verificados.

B.1. Ordem de disposição geral

- Todas as vias devem ser dispostas anteriormente ao início da disposição das vagas. A ordem de disposição de vias e vagas influencia significativamente na resolução do problema, e a distribuição de vias anterior à distribuição de vagas leva a resultados melhores, além de serem obtidos de forma mais simples.

B.2. Disposição das vias

- As vias devem ser dispostas de forma que seu traçado seja o menor possível, o que pode ser obtido locando-as próximas do centro da sub-região retangular do estacionamento (transversalmente) a que pertencem, paralelas ao comprimento desta mesma sub-região. Vias menores são desejáveis por diminuírem o tempo necessário às manobras de estacionamento.
- A disposição no sentido transversal deve respeitar os tipos permitidos de disposição de vagas. O posicionamento transversal ideal será diferente para cada possibilidade de distribuição (vagas paralelas à via, vagas perpendiculares à via ou vagas bloqueadas).
- Os obstáculos menores também devem ser considerados na definição da posição transversal ótima da via (determinada pela posição transversal que leva a um maior número total de vagas). Devem ser adotadas alternativas em nível decrescente de qualidade, caso algum obstáculo impossibilite a disposição ótima da via.
- Todas as vias devem estar interconectadas, de forma que a partir de qualquer uma delas seja possível chegar à porta do estacionamento.

B.3. Disposição das vagas

- As vagas devem ser dispostas primeiramente nas vias principais (aquelas que geram maior número total de vagas) e em seguida nas vias de menor importância. Esta ordem leva, em geral, a melhores disposições.
- A ordem de disposição dos tipos de vaga deve ser o seguinte: vagas perpendiculares à via, vagas paralelas à via e finalmente preencher os espaços ainda livres com vagas bloqueadas. Vagas bloqueadas são aquelas que não possuem comunicação direta com as vias, mas sim com uma outra vaga que tem acesso direto à via. Para que o veículo estacionado em uma vaga bloqueada saia é necessário que a vaga não bloqueada adjacente esteja vazia.

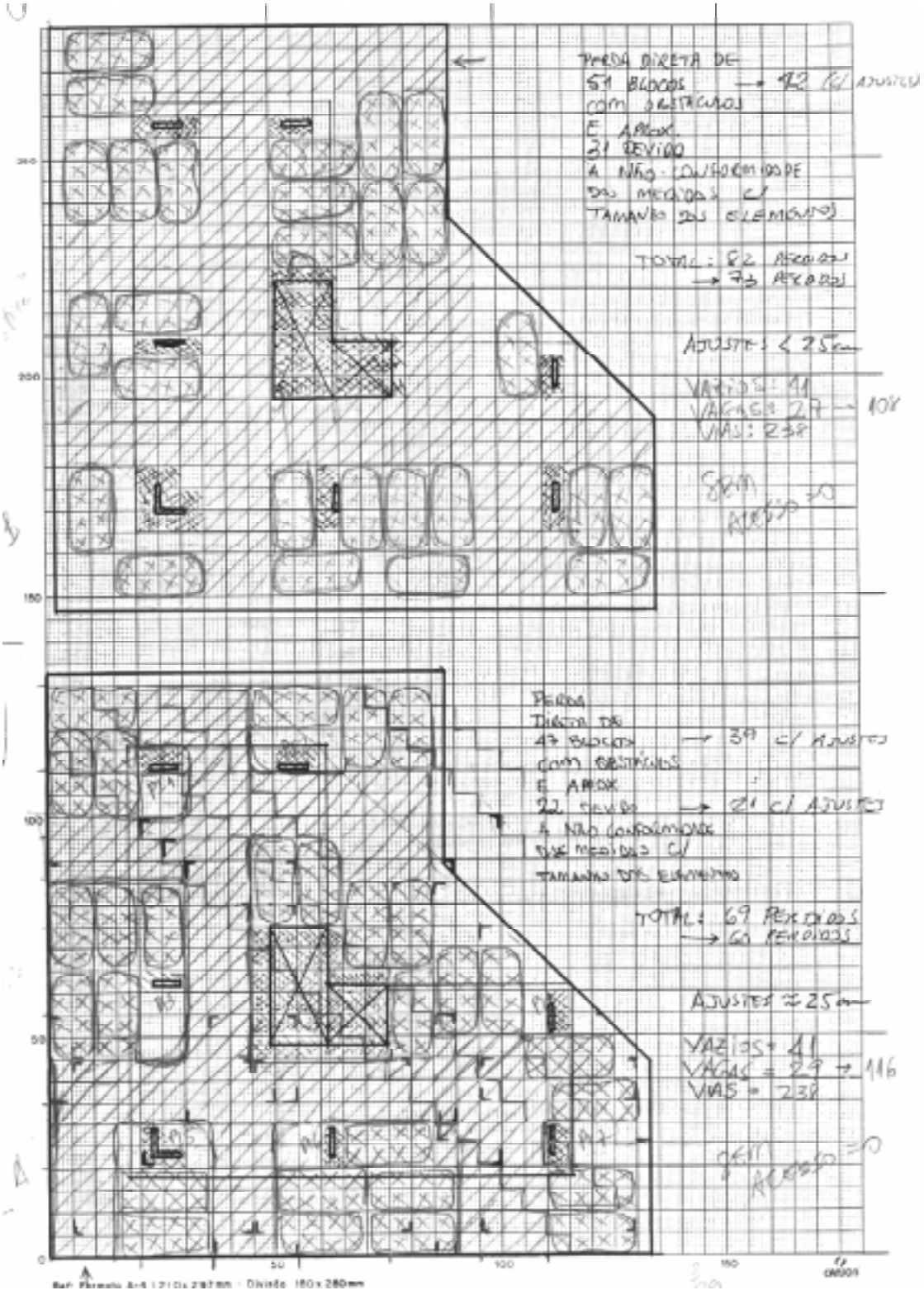


Figura B.1 – Estudo de disposição 1

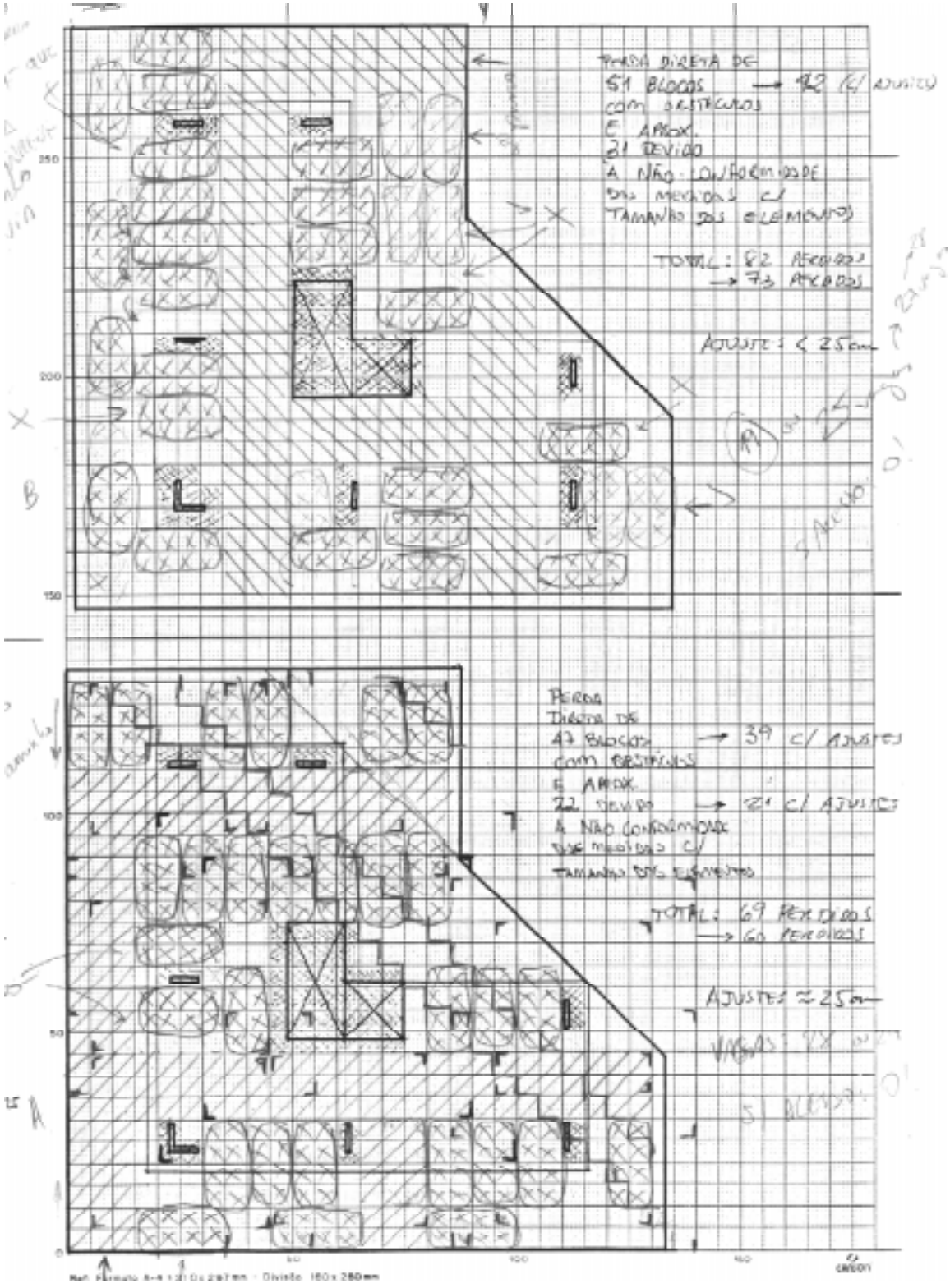


Figura B.2 - Estudo de disposição 2

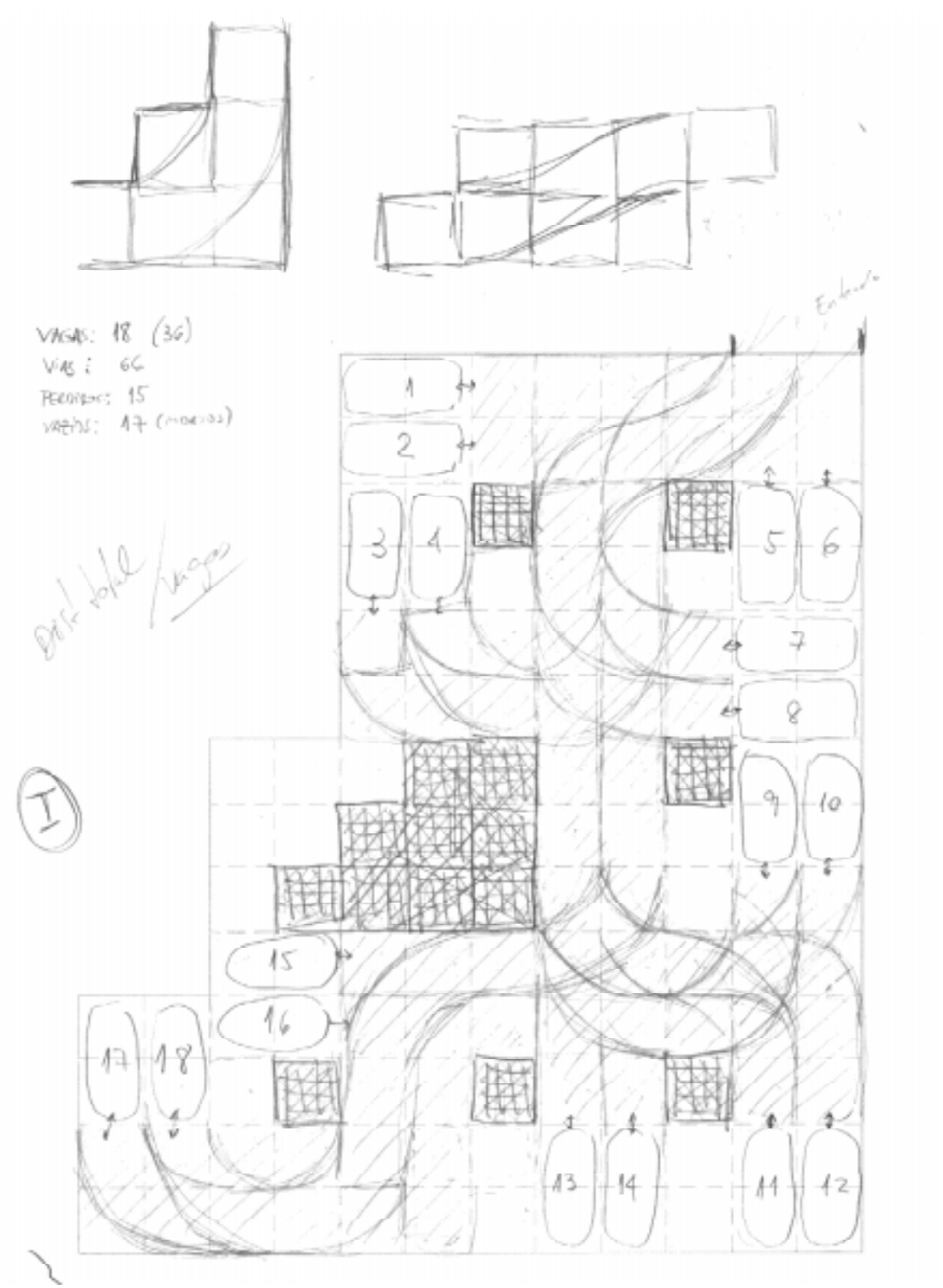


Figura B.3 - Estudo de disposição 3

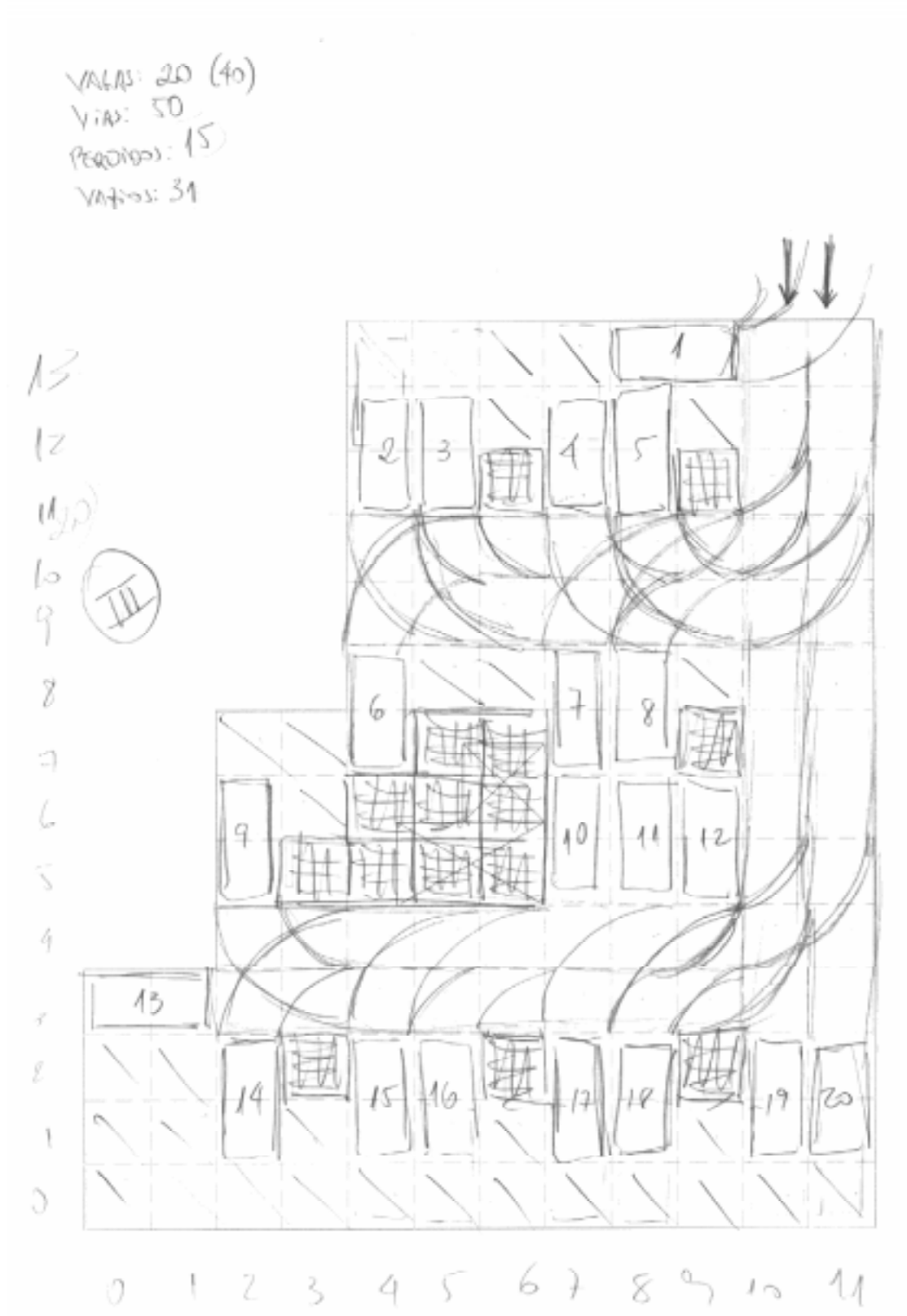


Figura B.4 - Estudo de disposição 4

Anexo C

Headers das Principais Classes

A seguir está o header (comentado) das principais classes que descrevem o problema estudado.

```
/*************************************************************************
                              Data Types
*************************************************************************/

// Kind types
// "Empty" spaces
#define KIND_EMPTY      0x00
#define KIND_DOOR       0x01

// "Occupied" spaces
#define KIND_SIDESTALL  0x80
#define KIND_FRONTSTALL 0x81
#define KIND_BLOCKSTALL 0x82
#define KIND_PATH       0x83
#define KIND_RELOCATEDPATH       0x84
#define KIND_OBSTACLE   0x85
// Not available/transitional spaces
#define KIND_BORDER     0xFE
#define KIND_EXTERNAL   0xFF

// Displacement Patterns Defines
#define DKIND_PATH              1
#define DKIND_SIDEVEICLE        2
#define DKIND_FRONTVEICLE       3
#define DKIND_BLOCKVEICLE       4

typedef void * CONNECT;

typedef class path * PPATH;


/*************************************************************************
                              Classes
*************************************************************************/


/*-----------------------------------------------------------------------*/
/*                              Garage Class                             */
/*                                                                       */
/* This class describes the garage. The solutions will be provided as    */
/* garage objects.                                                       */
/*-----------------------------------------------------------------------*/

class garage
{
private:
   POLYLINE * geometry;                     // Pointer to geometry (polyline)
   UCHAR discrete[MATRIX_YSIZE][MATRIX_XSIZE];  // Matrix where the objects will be placed
   UINT  block_size;                        // Size used blocks on matrix (cm)
   LINE * garagedoor;                       // Pointer to wall that contains the door.
   class access * theaccess;                // Pointer to the access object
   class veicle * standardveicle;           // Standard Veicle: source of basic measures
   class obstacle * obstacles;              // Pointer to the first obstacles (linked list)

   // Pattern distribution data - List the pattern numbers ordered by "value".
   // Greater value means the transversal distribution is better.
   // Vector positions: 0- "value" 1- Total transversal size 2- Distribution Number
   INT patternorder[MAXDISTPATTERNS][3];

   // Organize stall/path displacement patterns in "best to worst" order.
   VOID OrganizeDisplacementPatterns(VOID);

   // Used to draw a line on the discrete matrix.
   // line is a pointer to a line; kind is the value to fill the matrix
   VOID PlaceLineOnMatrix (LINE * line, UCHAR kind);
```

```
   // Used to define the internal area of the garage on the matrix
   // (It starts the use of MatrixFill)
   INT DefineInternalOnMatrix(VOID);

   // Recursive Routine to fill a Raster area
   // x,y is the initial point; value is the value to be used on the fill process.
   VOID MatrixFill (INT x, INT y, UCHAR value);

public:
   // Constructor.
   // points is a vector of points that will be used to generate the geometry polyline
   // door ia the number of the segment that will be the door
   // blocksize is the size of the blocks of the matrix
   // obspoints is a vector of points that will be used to generate the obstacle objects
   // sdtveicle is a pointer to the veicle object source of the measures used on this garage
   garage(INT points[MAXWALLPOINTS][2], INT door, UINT blocksize, INT obspoints[MAXOBSTACLEPOINTS][2],
class veicle * stdveicle);

   // Destructor
   ~garage();

   // COPY and ASSIGNMENT NOT DEFINED. IF NEEDED, THEY MUST BE CREATED

   // This Function plots the garage on the screen (STDOUT)
   VOID ShowASCIIGarage(VOID);
};


/*--------------------------------------------------------------------*/
/*                            Veicle Class                            */

/*                                                                    */
/* This class describes attributes and methods that are important in a  */
/* veicle so others objects can define their sizes                    */
/*--------------------------------------------------------------------*/

class veicle
{
private:
   UINT width;          // the widht of the veicle, in cm
   UINT lenght;         // the lenght of the veicle, in cm
   UINT extra_space;    // Additional space to extra movements, in cm
   UINT height;         // the height of the veicle, in cm (*not used*)
   UINT curve_diameter; // The minimum external diameter so the veicle
                        // can turn 90 degrees, in cm. (*not used*)
   UINT speed;          // the expected velocity, in cm/s (*not used*)

public:
   // Constructors
   // _width, _lenght and _height are the veicle measures (cm, height not used)
   // _extra_space is additional space to allow maneuvers, etc (cm)
   // _curve_diameter is a geometrical parameter that depends on veicle mechanics (cm, not used)
   // _speed is the speed of the veicle (cm/s, not used)
   veicle(UINT _width, UINT _lenght,  UINT _height, UINT _extra_space, UINT _curve_diameter, UINT
_speed);

   // _width, _lenght are the veicle measures (cm)
   // _extra_space is additional space to allow maneuvers, etc (cm)
   // _curve_diameter is a geometrical parameter that depends on veicle mechanics (cm, not used)
   veicle(UINT _width, UINT _lenght, UINT _extra_space, UINT _curve_diameter);

   // _width, _lenght are the veicle measures (cm)
   veicle(UINT _width, UINT _lenght);

   // Destructor
   ~veicle();

   // COPY and ASSIGNMENT NOT DEFINED. IF NEEDED, THEY MUST BE CREATED

   // Show object data (on STDOUT)
   VOID ShowInfo();

   // Friend Classes Declarations
   // Since this class values are used to almost all of implemented classes,
   // they are all declared as friends. It's up to them do not touch the values
   // inside this class.
   friend class stallbase;    // Stall uses veicle objects to set-up
   friend class stall;        // Stall uses veicle objects on some operations
   friend class access;       // Access uses veicle objects to set-up
   friend class pathbase;     // Pathbase uses veicle objects to set-up
   friend class path;         // Path uses veicle objects on some operations
};
```

```
/*-------------------------------------------------------------------------*/
/*                           Stall Base Class                              */
/*                                                                         */
/* This class describes attributes and methods that are important in a     */
/* stall size.                                                             */
/*-------------------------------------------------------------------------*/

class stallbase
{
protected:
   UINT width;          // the widht of the stall, in cm
   UINT lenght;         // the lenght of the stall, in cm
   UCHAR * matrix;      // Matrix where the stall is placed.
   UINT blocksize;      // size of block

public:

   // Constructor
   // standardveicle is a pointer to the design veicle object being used
   // matrix is a pointer to the discrete matrix being used
   // blocksize is the size of the blocks on the discrete matrix
   stallbase(class veicle * standardveicle, UCHAR * matrix, UINT blocksize);

   // Destructor
   ~stallbase();

   // COPY and ASSIGNMENT NOT DEFINED. IF NEEDED, THEY MUST BE CREATED

   // Returns the Stall Width in blocks if blocksize!=0. If blocksize=0
   // it returns the "real" value of the width.
   UINT GetWidth(UINT blocksize);

   // Returns the Stall Lenght in blocks if blocksize!=0. If blocksize=0
   // it returns the "real" value of the Lenght.

   UINT GetLenght(UINT blocksize);
};


/*-------------------------------------------------------------------------*/
/*                             Stall Class                                 */
/*                                                                         */
/* This class describes attributes and methods that are important in a     */
/* stall so the car displacement can be done.                              */
/*-------------------------------------------------------------------------*/

class stall : public stallbase
{
private:
   /* Geometry */
   UINT position;             // Stall relative to path position (frontal/parallel/blocked)
   UINT side;                 // Left or Right of the path...
   RECTANGLE * area;          // Area ocupied by the stall
   POINT * nextpos;           // Point to next stall in the same side
   stall * nextstall;         // pointer to next stall
   stall * prevstall;         // pointer to previous stall
   stall * unlinkedstall;     // blocked stall created but initially not used
   POINT * backpos;           // Point to back stall in the same side

   // This function enable or destroy secondary stalls after all other stalls are
   // placed. If the place is still empty, it adds the stall to the stall linked
   // list. If not, it destroys the blocked stall object.
   VOID LinkAndShowSecondaryStall(VOID);        // Links and Show a secondary stall

   // Draw a Stall on the matrix
   VOID DrawOnMatrix(VOID);

public:

   // Constructor
   // x,y is the position where the stall will be created. This position must be over the path axis!
   // maxx, maxy are the end points of this path
   // side is the side where the stall is being created
   // position is the relative position of the stall (parallel, perpendicular, blocked)
   // standardveicle is the pointer to the design veicle object
   // Matrix is the discrete matrix where the stall will be placed
   // blocksize is the size of the blocks on the matrix
   // nextstpos is a pointer to a point, where the next stall position will be returned by this
function
   stall(INT x, INT y, INT maxx, INT maxy, UINT side, UINT position, UINT pathwidth, class veicle *
standardveicle, UCHAR * matrix, UINT blocksize, POINT * nextstpos);
```

```
   // Destructor
   ~stall();

   // COPY and ASSIGNMENT NOT DEFINED. IF NEEDED, THEY MUST BE CREATED

   // This function returns the pointer to the next stall
   stall * GetNext(VOID);

   // This function returns the pointer to the previous stall
   stall * GetPrevious(VOID);

   // This function redefines the pointer that points to the next stall
   VOID SetNext(stall * nstall);

   // This function redefines the pointer that points to the previous stall
   VOID SetPrevious(stall * pstall);

   // Show geometrical Stall data on STDOUT
   VOID ShowInfo (VOID);

   // The class path uses several functions of this class, so this was
   // declared as friend.
   friend class path;
};


/*---------------------------------------------------------------------*/
/*                          Path Base Class                            */
/*                                                                     */
/* This class describes basic attributes and methods of a path         */
/*---------------------------------------------------------------------*/

class pathbase
{
protected:
   UINT width;               // the widht of the path, in cm
   UCHAR * matrix;           // Pointer to the displacement matrix
   class veicle * stdveicle; // standard veicle for this path
   UINT blocksize;           // Blocksize for this path

public:
   // Constructor
   // standardveicle is the pointer to the design veicle object
   // matrix is the discrete matrix where the path will be placed
   // blocksize is the size of the blocks of the matrix
   pathbase(class veicle * standardveicle, UCHAR * matrix, UINT blocksize);

   // Destructor
   ~pathbase();           // Destructor

   // COPY and ASSIGNMENT NOT DEFINED. IF NEEDED, THEY MUST BE CREATED

   // Returns the Path Width in blocks if blocksize!=0. If blocksize=0
   // it returns the "real" value of the width.
   UINT GetWidth(UINT blocksize);
};


/*---------------------------------------------------------------------*/
/*                            Path Class                               */
/*                                                                     */
/* This class describes attributes and methods of a path               */
/*---------------------------------------------------------------------*/

class path : public pathbase
{
private:
   INT distribution;         // Type of distribution (number of the distribution pattern)
   INT distvalue;            // Value associated to the distribution type
   INT relocatedpath;        // Informs if this path was ideal or relocated
   RECTANGLE * basearea;     // Rectangle of area actualy used by the path
   RECTANGLE * influence;    // Influence rectangle (the rectangle that generated this path
   stall * stalls;           // Pointer to the first stall of the path
   INT stalldelta[4];        // space used by stalls on both sides (0-left 1-right)
   LINE * baseline;          // the line that describes the axis of the path segment
   class path * prevpath;    // Pointer to the previous path
   class path * nextpath;    // Pointer to the next path

   // Tree Organization attibutes
   class path * parentpath;  // Pointer to the parent path
   INT treelevel;            // level of path in the tree
```

```
   // This function draws the path on the matrix
   VOID DrawOnMatrix(VOID);

   // Function that places the stalls along this path
   VOID PlaceStalls(UINT mode);

   // This function starts the stall drawings on the matrix
   VOID DrawStallsOnMatrix(VOID);

   // Link and Show the Secondary Stalls associated to this path (blocked stalls)
   VOID LinkAndShowSecondaryStalls(VOID);

   // Returns if the space for this path is not occupied on matrix
   INT IsFreeOnMatrix(VOID);

   // Calculates the area occupied by the path
   VOID BaseAreaCalc(VOID);
public:
   // constructor
   // basewall is a pointer to the line of a wall that defines a rectangle
   // distance is the second value used to determine the influence rectangle
   // patternorder is the organized distribution patterns that will be used to determine the path
position
   // stdveicle is the design veicle object
   // matrix is the workarea matrix
   // blocksize is the size of the matrix blocks
   path (LINE * basewall, INT distance, INT patternorder[][3], class veicle * stdveicle, UCHAR *
matrix, INT blocksize);

   // Destructor
   ~path();

   // COPY and ASSIGNMENT NOT DEFINED. IF NEEDED, THEY MUST BE CREATED

   // This function returns the pointer to the next path segment
   class path * GetNext(VOID);

   // This function returns the pointer to the previous segment
   class path * GetPrevious(VOID);

   // This function changes the pointer that points to the next path segment
   VOID SetNext(class path * npath);


   // This function returns the pointer that points to the previous segment
   VOID SetPrevious(class path * ppath);

   // Return the parent path segment of this path, after the path-tree has been generated
   class path * GetTreeParent(VOID);

   // Changes the parent path segment of this path segment.
   VOID SetTreeParent(class path * ppath);

   // Returns the tree level of this path segment
   INT GetTreeLevel(VOID);

   // Changes the tree level of this path segment
   VOID SetTreeLevel(INT level);

   // Show geometrical path data on STDOUT
   VOID ShowInfo(VOID);

   // This function returns the number of the distribution associated to this path
   INT GetDistribution(VOID);

   // This function returns the lenght of this path (real value)
   UINT GetLenght(VOID);

   // Return the "value" of this path distribution, based on distribution and lenght
   INT GetValue(VOID);

   // Returns the space used by Stalls in the side of point
   INT GetUsedSpace(DJ_point * point);

   // Since Access class usses lots of data of paths, it was declared as friend.
   friend class access;
};


typedef class path PATH;
```

```
/*-------------------------------------------------------------------------*/
/*                            Access Class                                 */
/*                                                                         */
/* This class describes attributes and methods of the access               */
/*-------------------------------------------------------------------------*/

class access
{
private:
   PATH * paths;       // Pointer to the first segment of the path
   INT blocksize;       // default Blocksize for this path
   UCHAR * matrix;      // Matrix for displacement

   // This function reorder the paths by lenght, smaller first.
   VOID OrderByPathLenghtUp(VOID);

   // This function reorder the paths by influence area, bigger first.
   VOID OrderByInfluenceAreaDown(VOID);

   // This function exchange two paths order in the double-linked list.
   VOID ExchangePathOrder(class path * path1, class path * path2);

   // Verifies if there is any path crossing a given path between its start and
   //  the point it crosses crosspath.
   INT CrossInStartOfPath(path * basepath, path * crosspath);

   // Verifies if there is any path crossing a given path between its end and
   //  the point it crosses crosspath.
   INT CrossInEndOfPath(path * basepath, path * crosspath);

   // This function delete all paths that has influence areas inside other
   // bigger ingluence area.
   VOID KillRedundantInfluencePaths(VOID);

   // This function deletes redundant small pieces of paths (end-of-path)
   VOID KillRedundantSmallPaths(veicle * stdveicle);

   // This function deletes paths that are parallel, serves almost the same area and
   // are too close
   VOID KillRedundantParallelPaths(veicle * stdveicle);

   // Draws the access  on the matrix
   VOID DrawOnMatrix(VOID);

   // Show geometrical path data on STDOUT
   VOID ShowPathsInfo(VOID);

   // Organizes path segments into a tree, returns the first path segment that
   // was not connected to other paths in the tree.
   class path * MakeTreeWithPaths(class DJ_line * door);



public:
   // constructor
   // blocksize is the size of the matrix block
   // geometry is a polyline describing the lines that may generate paths
   // door is the pointer to the line that describes the door
   // patternorder is the organized distribution patterns that will be used to determine the path
position
   // stdveicle is the design veicle object
   // matrix is the workarea matrix
   access(INT blocksize, POLYLINE * geometry, LINE * door, INT patternorder[][3],class veicle *
stdveicle, UCHAR * matrix);

   // Destructor
   ~access();

   // COPY and ASSIGNMENT NOT DEFINED. IF NEEDED, THEY MUST BE CREATED

   // Function that place stalls along the paths...
   VOID PlaceAndShowStalls(UINT mode);

   // Link and Show the Secondary Stalls associated to this access
   VOID LinkAndShowSecondaryStalls(VOID);
};
```

```
/*--------------------------------------------------------------------*/
/*                            Obstacle Class                          */
/*                                                                    */
/* This class describes attributes and methods that are important in a  */
/* obstacle so the car displacement can be done.                      */
/*--------------------------------------------------------------------*/

class obstacle
{
private:
   POLYLINE * geometry;              // the obstacle geometry
   obstacle * nextobstacle;     // pointer to next obstacle
   obstacle * prevobstacle;     // Pointer to previous obstacle
   UINT blocksize;
   UCHAR * matrix;              // Matrix where the obstacle is drawn.

public:
   // Constructor
   // obspoints is a vector of points that will be used to generate the obstacle geometry
   // matrix is a pointer to the discrete matrix being used
   // blocksize is the size of the blocks on the discrete matrix
   obstacle(INT obspoints[MAXOBSTACLEPOINTS][2], UCHAR * matrix, UINT blocksize);

   // Destructor
   ~obstacle();

   // COPY and ASSIGNMENT NOT DEFINED. IF NEEDED, THEY MUST BE CREATED

   // This function returns the pointer to the next obstacle
   obstacle * GetNext(VOID);

   // This function returns the pointer to the previous obstacle
   obstacle * GetPrevious(VOID);

   // This function changes the pointer that points to the next obstacle
   VOID SetNext(obstacle * nobstacle);

   // This function returns the pointer that points to the previous obstacle
   VOID SetPrevious(obstacle * pobstacle);

   // Draw the obstacle on the matrix
   VOID DrawOnMatrix(VOID);
};
```

Anexo D

Instruções de uso do software desenvolvido

D.1. Apresentação

Sendo um dos objetivos desta pesquisa, o software em questão tem por função desenvolver uma solução para um estacionamento, ou seja, dispor as vias e vagas numa determinada área, considerando todas as interferências existentes.

Para que ele possa desempenhar esse papel, alguns dados de entrada são necessários. Estes dados devem respeitar algumas regras, sob pena do programa não funcionar corretamente. A checagem de erros executada por ele nos arquivos de entrada é mínima e não cobre grande parte dos erros possíveis. Portanto, é preciso grande cuidado na entrada de dados.

Em alguns casos será necessária alguma complementação manual (vias não conectadas, nenhuma via ligada à porta, etc.). Isso não ocorrerá na maioria dos casos, entretanto. Extensões ao programa serão necessárias para que ele execute essas atividades automaticamente.

D.2. Operação

O programa foi projetado para operar por linha de comandos. Para sua execução é necessária a utilização de algum sistema Win32 compatível (Windows 95/98/Me/NT/2000/XP). Sua execução deve ser chamada pelo *prompt de comandos* e a sintaxe correta é:

`garage.exe` ***`[arquivo_de_dados]`***

Onde *[arquivo_de_dados]* é um parâmetro opcional que especifica os arquivos de dados. Caso seja omitido, o nome *default* será utilizado. Este nome é um radical que será usado para construir o nome de dois arquivos de dados a serem lidos: o arquivo de definição de área interna (.WAL) e o arquivo de definição dos obstáculos (.OBS). Assim, os arquivos de entrada padrão são: *default.wal* e *defaul.obs*. Se for fornecido o nome *projeto* como parâmetro para o programa, os arquivos *projeto.wal* e *projeto.obs* serão utilizados.

O arquivo com extensão .OBS não é obrigatório. Caso ele não exista, a garagem será considerada sem obstáculos menores. O arquivo com extensão .WAL é obrigatório, no entanto. Sem sua existência, um erro será reportado (A descrição da estruturação destes arquivos, bem como arquivos de dados exemplo estão disponíveis nas Seções D.3 e D.4).

A saída do programa é feita na tela (STDOUT). Caso se deseje a saída para um arquivo, basta utilizar a seguinte linha de comando:

garage.exe *[arquivo_de_dados]* > *arquivo_de_saida*

Isso fará com que os dados sejam colocados dentro do arquivo *arquivo_de_saida*. Outra opção é ter a saída diretamente na impressora. Para isso basta usar a seguinte linha de comando:

garage.exe *[arquivo_de_dados]* > PRN

A saída gerada pelo programa terá o seguinte aspecto:

```
*** Stall Displacement Solver v1.0.0 for Win32


 Creating Garage for the following veicle:

Veicle:
Width:         184cm
Lenght:        450cm
Height:        150cm
ExtraSpace:    10cm
CurveDiameter: 1130cm
Speed:         85cm/s


Blocksize Used: 225


The Geometry was generated.
The Door was generated.
Creating Obstacles...
The Matrix was generated.
Displacement Patterns Organized.


Generating Paths...

Deleting redundant parallel paths...
Deleting redundant area paths...
Deleting redundant small paths...
Deleting redundant parallel paths...
Ordering by size...
Generating Path Tree...
Maximum level of this path: 2
Ordering by Influence Area...


Creating Primary and Unlinked Secondary Perpendicular Stalls...
Placing Stalls...

Creating Primary Parallel Stalls...
Placing Stalls...

Linking and Placing Secondary Perpendicular Stalls...
Placing Stalls...

Paths Created
```

```
: : : : : : : : : : : : : : : : : : : :
: : : : : : : : : : : : : : : : : : : :
: : : : : : : : : : : : : : : : : : : :
: : : : : : : : : : : : : : : : : : : :
: : : : : : : : : : : : : : : : : : : :
: : : : : : : : : : : : : : : : : : : :
: : : : D F F * * F F D : : : : : : : :
: : : :   F F * * F F   : : : : : : : :
: : : :     O * * O     : : : : : : : :
: : : :   F F * * F F   : : : : : : : :
: : : :   F F * * F F   : : : : : : : :
: : : :   F F * * F F   : : : : : : : :
: :       O O * * O     : : : : : : : :
: :     O O O * * F F   : : : : : : : :
: : B O O O O * * F F   : : : : : : : :
: : B B B F F * * F F   : : : : : : : :
F F F   F F F * * F F   : : : : : : : :
F F F O F   O * * O     : : : : : : : :
* * * * * * * * * F F   : : : : : : : :
* * * * * * * * * F F   : : : : : : : :

Symbols:
  D - Door (may be covered)
  P - Stall Parallel to the Path
  F - Stall Frontal to the Path
  B - Stall Blocked by another stall
  + - Path placed on the optmal place
  * - Relocated Path
  O - Obstacle
  : - External area
```

Primeiramente ele exibe a versão do programa, seguido por uma descrição do veículo de projeto, utilizado para a resolução. O tamanho do bloco usado na solução também é mostrado (valor em centímetros). Quanto menor o tamanho do bloco, mais refinada a solução.

Em seguida são exibidas informações para que se acompanhe o desenvolvimento da solução (cada operação do procedimento de disposição de vagas). Finalmente é apresentada a solução em ASCII da área fornecida (equivalente à da figura 5.3.2), com a respectiva legenda.

D.3. Arquivos de Dados

Como descrito anteriormente, há dois tipos de arquivos de dados, existindo também um arquivo de configuração geral do programa. Cada um deles será descrito (seu formato e como preencher dados nos mesmos) nos itens seguintes.

*D.3.1 O arquivo de configuração **garage.cfg***

O arquivo ***garage.cfg*** oferece a possibilidade de configurações gerais do programa. A estrutura deste arquivo é a seguinte:

```
Garage Config 0.2
0
```

A linha 1 é uma string que descreve a versão do arquivo de configuração, e deve ser exatamente igual à indicada acima. Qualquer diferença fará com que o programa não funcione.

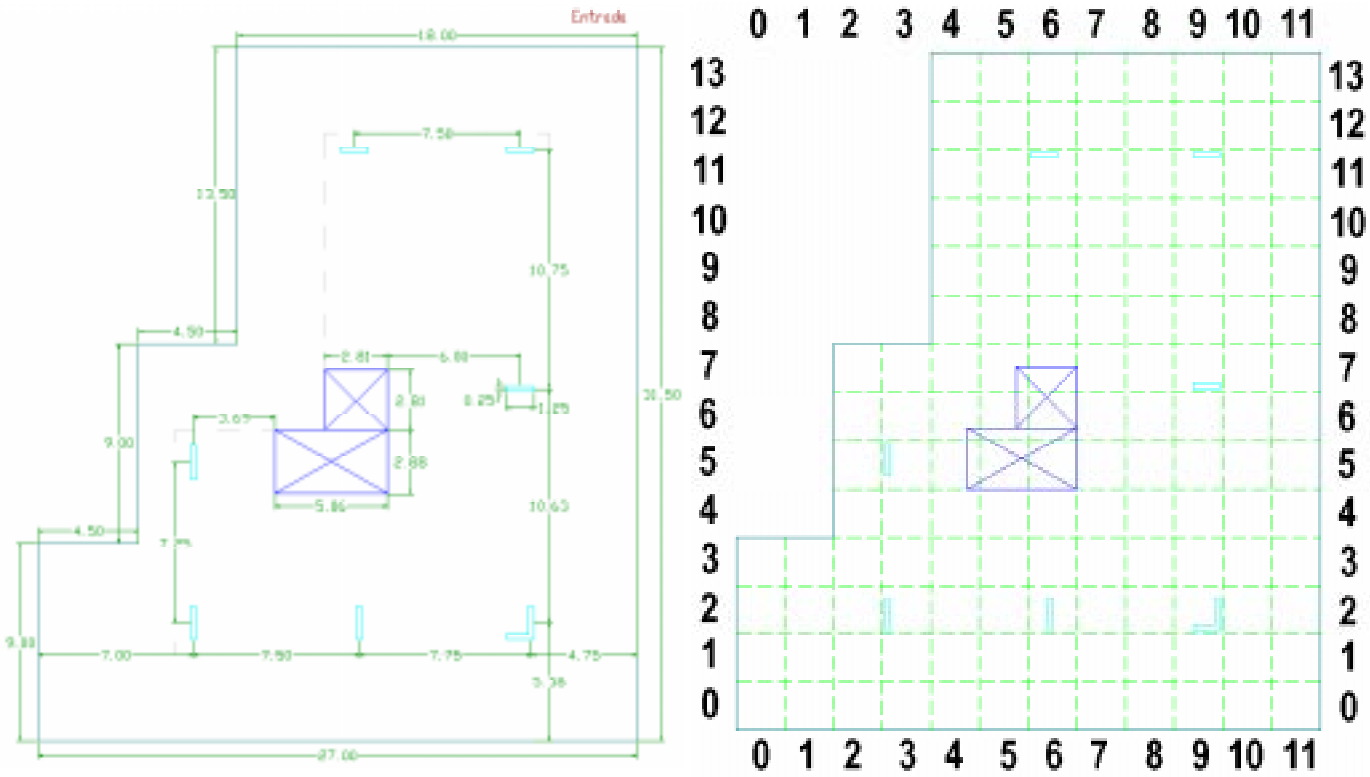
A linha 2 indica qual o tipo de distribuição a ser usada. O tipo de distribuição é o que indica para o programa se ele irá tentar otimizar os espaços para a existência de vagas bloqueadas ou não. O valor 0 nesta linha indica para **não considerar** vagas duplas na disposição das vias e o valor 1 indica para **considerar** as vagas duplas. Note que nem sempre as soluções são diferentes e é preciso utilizar as duas formas para podermos avaliar qual distribuição é mais satisfatória.

D.3.2 O arquivo de definição de área livre ***default.wal***

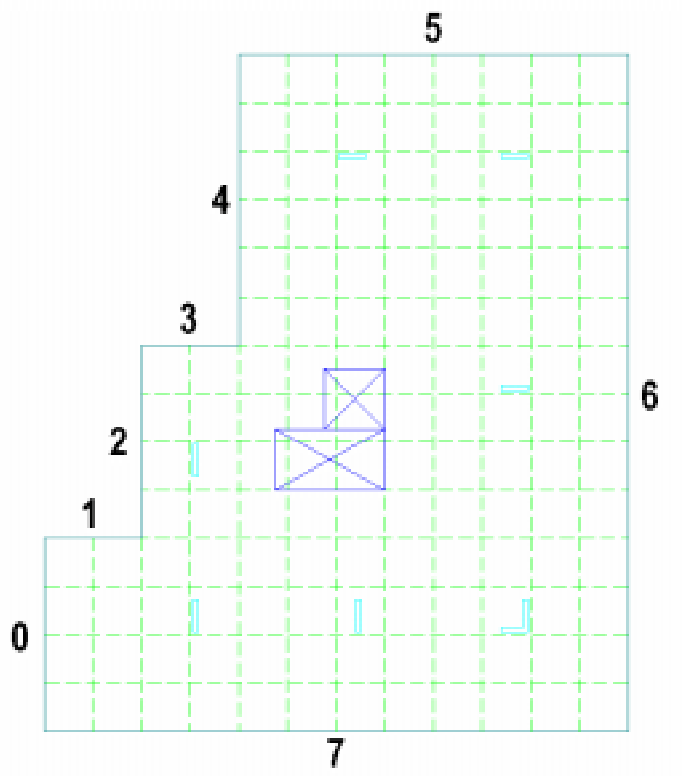
O arquivo ***default.wal*** (ou *nome_fornecido.wal*) serve para determinar a geometria da área livre interna, onde serão distribuídos os elementos de garagem. A primeira linha deve conter uma string de versão do arquivo de dados conforme indicado abaixo.

```
Garage Walls 0.2
```

Qualquer diferença fará com que o programa deixe de funcionar. A segunda linha indica qual dos segmentos que compõem a parede contém a porta de entrada. Para a obtenção deste valor, o primeiro passo é quadricular a planta como indicado, com elementos de mesmo tamanho:



Feito isso, escolhe-se um elemento de um dos cantos da figura como referência (aqui adotado o elemento 0,0) e numera-se as paredes no sentido horário, começando pelo valor 0, como pode ser observado na figura ao lado.

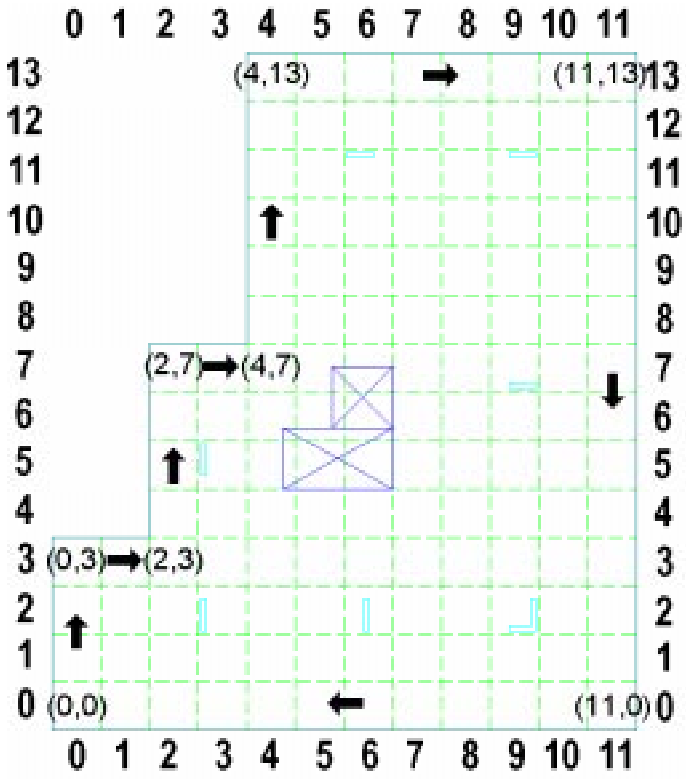


Portanto, a parede número 5 é a que contém a porta. Indicamos este valor na segunda linha. Na terceira linha indicamos qual é o tamanho dos elementos do quadriculado. Neste caso, adotamos 225 cm. O arquivo .WAL até este ponto seria assim:

```
Garage Walls 0.2
5
225
```

Note que ao quadricular a área deve-se tomar alguns cuidados. Todos os elementos devem pertencer ao primeiro quadrante, ou seja, ao numerar as linhas e colunas (da forma indicada nas figuras) **não podem existir linhas ou colunas negativas**. Além disso, o tamanho dos elementos indicado na terceira linha deve ser tal que o número de linhas e colunas não ultrapasse 20 (em outras palavras, os números de linhas e colunas devem variar de 0 a 19). Esta limitação foi utilizada por atender às necessidades da presente pesquisa, com elementos de 225 cm.

A partir da quarta linha devem ser indicados os elementos de canto da área interna da figura, um por linha, partindo do elemento de referência (o mesmo usado para numerar as paredes, que neste exemplo foi o elemento 0,0) e percorrendo os elementos de canto das paredes no sentido horário, como na figura ao lado, de forma que eles descrevam segmentos cujo lado direito indica sempre a área interna do estacionamento.



Ou seja, a partir da quarta linha teremos os elementos na ordem indicada na figura, um por linha (0,0; 0,3; 2,3; ...; 11,13; 11,0; 0,0).

Ao fim da seqüência devemos usar o valor –1,-1 para indicar o fim da figura. O conteúdo do arquivo .WAL completo fica assim:

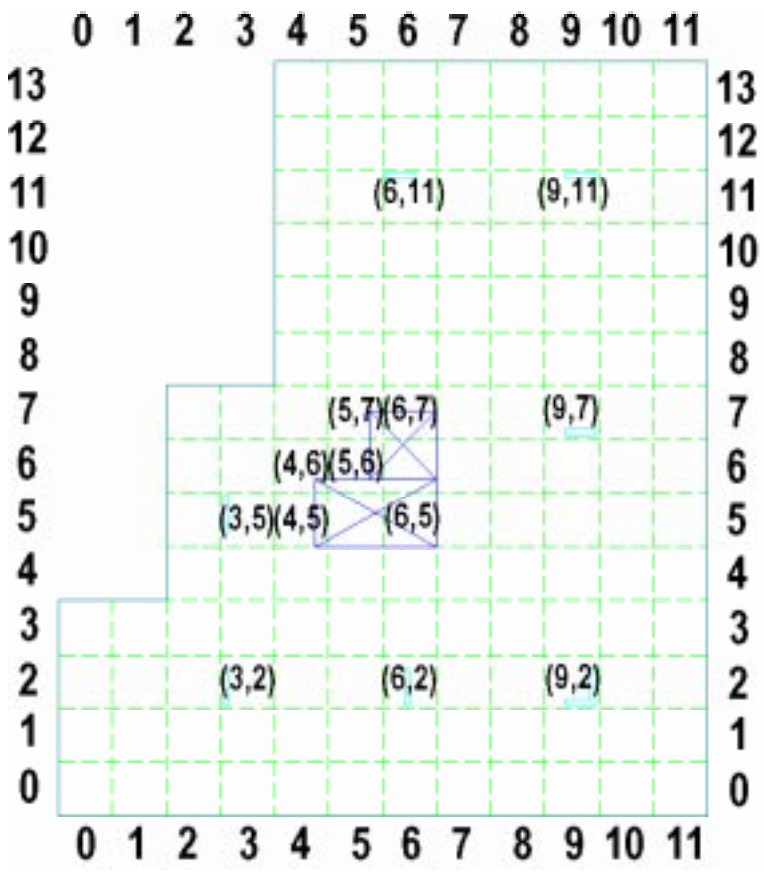
```
Garage Walls 0.2
5
225
0,0
0,3
2,3
2,7
4,7
4,13
11,13
11,0
0,0
-1,-1
```

Aqui também existem algumas observações a serem feitas. Os elementos indicados serão usados, dois a dois, para descreverem a geometria da área de estacionamento. Esses elementos **devem** descrever uma área com lados ortogonais uns aos outros. Caso contrário o programa não funcionará corretamente. Pelo mesmo motivo, o elemento inicial e o final devem ser iguais, indicando uma geometria fechada.

D.3.3 O arquivo de definição de obstáculos ***default.obs***

O arquivo ***default.obs*** (ou *nome_fornecido.obs*) oferece a possibilidade de determinar a geometria de diversos obstáculos na área do estacionamento. O arquivo contém na primeira linha a descrição da versão, e a partir da segunda contem os elementos de canto da geometria de cada obstáculo, seguindo as mesmas regras usadas para a descrição da área interna do estacionamento.

O fim da seqüência de elementos de canto de cada obstáculos deve indicado por um valor -1,-1, e o final da seqüência de obstáculos deve ser indicado por uma dupla -1,-1 e -1,-1.



Assim, para a garagem de nosso exemplo, o arquivo de dados ***default.obs*** gerado seria

```
Garage Obstacles 0.2
3,2
3,2
-1,-1
6,2
6,2
-1,-1
9,2
9,2
-1,-1
9,7
9,7
-1,-1
9,11
9,11
-1,-1
6,11
6,11
-1,-1
3,5
3,5
-1,-1
4,5
4,6
5,6
5,7
6,7
6,5
4,5
-1,-1
-1,-1
```

D.4. Exemplos

Seguem abaixo alguns exemplos de arquivos .WAL e .OBS.

D.4.1. Garagem em L (sem obstáculos)

Arquivo ***gl.wal***:

```
Garage Walls 0.2
4
225
0,0
0,19
5,19
5,7
15,7
15,0
0,0
-1,-1
```

D.4.2. Garagem em E (sem obstáculos)

Arquivo ***ge.wal***:

```
Garage Walls 0.2
10
225
0,0
0,19
19,19
19,14
6,14
6,12
19,12
19,7
6,7
6,5
19,5
19,0
0,0
-1,-1
```

D.4.3. Garagem em S (sem obstáculos)

Arquivo ***gs.wal***:

```
Garage Walls 0.2
0
225
0,0
0,5
4,5
4,4
15,4
15,8
0,8
0,19
19,19
19,14
15,14
15,15
4,15
4,12
19,12
19,0
0,0
-1,-1
```

D.4.4. Garagem Genérica (sem obstáculos)

Arquivo ***ggs.wal***:

```
Garage Walls 0.2
11
500
2,0
2,6
0,6
0,8
2,8
2,12
5,12
5,6
7,6
7,4
5,4
5,0
2,0
-1,-1
```

D.4.5. Garagem Genérica (com obstáculos)

Arquivo ***ggc.wal***:

```
Garage Walls 0.2
11
225
0,0
0,15
15,15
15,5
9,5
9,9
5,9
5,7
3,7
3,3
5,3
5,0
0,0
-1,-1
```

Arquivo ***ggc.obs***:

```
Garage Obstacles 0.2
8,10
8,10
-1,-1
10,13
10,12
-1,-1
-1,-1
```