5

Path Finding

As in the real world, finding a path from one place to another is a common – if not the most common – algorithmic problem in computer games. Although the problem can seem fairly simple to us humans (most of the time), a surprising amount of the total computation time in many commercial computer games is spent in solving path-finding problems. The reasons for this are the ever-increasing complexity of game world environments and the number of entities that must be calculated. Moreover, if the environment changes dynamically (e.g. old paths become blocked and new ones are opened), routes cannot be solved beforehand but only reactively on the spot as the game progresses. Real-time interaction puts even further constraints, because the feedback to the human player should be almost instant and the path must be found before he gets too impatient to wait any longer.

The problem statement of path finding is simple: Given a start point s and a goal point r, find a path from s to r minimizing a given criterion. Usually this cost function is travelling time, which can depend on the distance, the type of terrain, or the mode of travel. We can think of path finding either as a search problem – find a path that minimizes the cost – or as an optimization problem – minimize the cost subject to the constraint of the path. Consequently, graph search methods can be seen as optimization methods, where the constraints are given implicitly in the form and weights of the graph. Although we can use general optimization methods such as simplex, they lose the graph-like qualities of the path-finding problem, which is why we focus mainly on the search problem throughout this chapter.

In an ideal case, we would do path finding in a continuous game world and solve for the route from s to r straightforwardly. Unfortunately, this is rarely a realistic option, since the search space gets too complex. Instead, we discretize the search space by restricting the possible *waypoints* into a finite set and reducing the paths to *connections* between them. In other words, we form a graph in which the vertices are the waypoints and the edges are the connections. We have thus reduced the original problem to that of finding a path in a graph (see Figure 5.1). The idea resembles travelling in the real world: Move to the closest waypoint (airport, bus stop, underground station, harbour etc.), go through waypoints until closest to the destination, exit the final waypoint, and proceed to the destination.

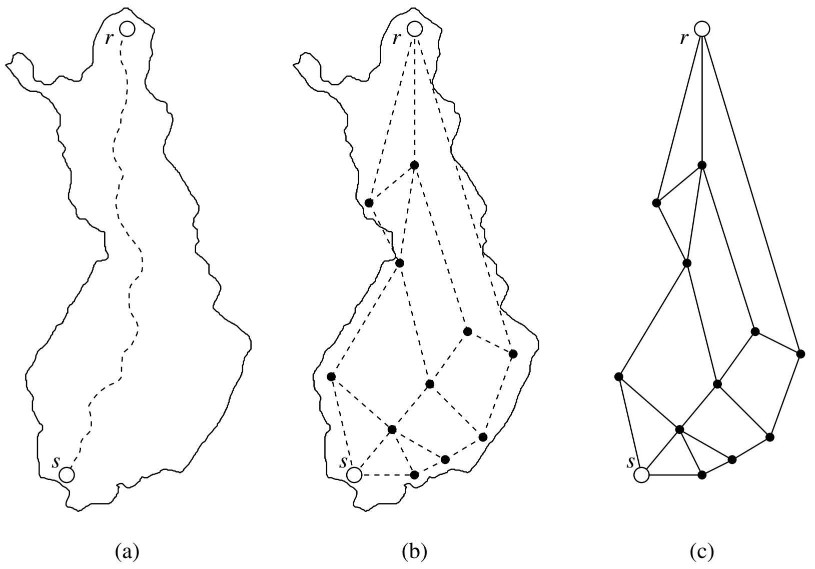

Figure 5.1 Real-world path finding is reduced into a graph problem by discretizing the search space into waypoints. The waypoints are the vertices and the connections between them are the edges of the graph.

This approach gives us a three-step method: First, we show how the game world can be discretized. On the basis of the discretization, we can form a graph, and the path-finding problem is transformed into that of finding the minimum path in the graph. Although there are several algorithms to solve this problem, we concentrate on A* algorithm, which uses a heuristic estimate function to enhance the search. Finally, when the minimum path in the graph has been found, it has to be realized as movements in the game world considering how realistic the movements look for the human observing them.

5.1 Discretization of the Game World

The first step in solving the path-finding problem in a continuous world is to discretize it. The type of the game world usually gives an indication on how this discretization ought to be done. We can immediately come up with intuitive choices for waypoints: doorways, centres of the room, along the walls, corners, and around the obstacles (Tozour 2003). Once the waypoints have been selected, we establish whether there is a connection between them based on the geometry of the game world. The connection can be associated with cost based on the distance or type of the environment, and this cost is set to be the weight of the edge.

Although the waypoints can be laid down manually during the level design, preferably it should be an automatic process. Two common approaches to achieve this are to superimpose a grid over the game world, or use a navigation mesh that observes the underlying geometry.

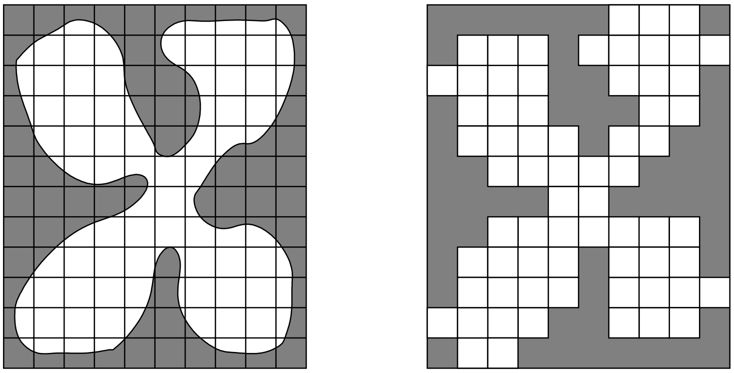

Figure 5.2 A square grid is laid over the game world. If the majority of the world inside a tile is open, the tile is included in the waypoints.

5.1.1 Grid

We can place a grid, which is a tiling of polygons (i.e. tessellation), over the game world. To simplify, we consider only grids in which each tile shares at most one edge with a neighbouring tile (see Figure 5.2). Now, the centre of a tile represents a waypoint, and its neighbourhood, composed of the adjacent tiles, forms the possible connections to other waypoints. The world inside the tile defines whether it is included in the waypoints and what are its connections to other waypoints.

Grids usually support random-access lookup, because each tile should be accessible in a constant time. The drawback of this approach is that a grid does not pay attention to the actual geometry of the game world. For instance, some parts of the world may get unconnected if the granularity of the grid is not fine enough. Also, storing the grid requires memory, but we can reduce this requirement, for example, by using hierarchical lookup tables (van der Sterren 2003).

There are exactly three regular tessellations composed of equilateral triangles, squares, or regular hexagons (see Figure 5.3). When we are defining a neighbourhood for triangular

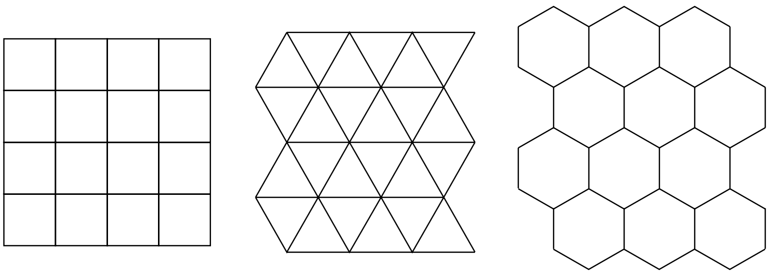

Figure 5.3 Square grid, triangular grid, and hexagonal grid are the only regular two-dimensional tessellations.

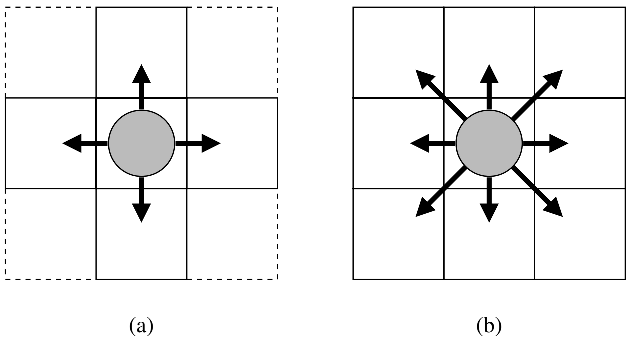

Figure 5.4 A square grid allows (a) four-connectivity and (b) eight-connectivity.

and square grids, we must first decide whether we consider only the tiles adjacent to the edges of a tile or also the tiles that share a corner point with the tile. Figure 5.4 illustrates the situation in a square grid: In the former case we have a four-connectivity (i.e. a tile has at most four neighbours), and in the latter case eight-connectivity. An obvious problem with eight-connectivity is that diagonal moves are longer than vertical or horizontal ones, which should be taken into account in calculations of distances. Because hexagonal grids allow only six-connectivity and the neighbours are equidistant, they are often used in strategy and role-playing games.

Instead of assigning the waypoints to the centre of the tiles, we can use the corners of the tiles. Now, the neighbourhood is determined along the edges and not over them. However, these two waypoint assignments are the dual of each other, since they can be converted to both directions. For the regular tessellations, the conversion is simple, because we can consider the centre of a tile as a corner point of the dual grid and vice versa, and – as we can see in Figure 5.3 – the square grid is the dual shape of itself and the triangular and hexagonal grids are the dual shapes of each other.

5.1.2 Navigation mesh

A navigation mesh is a convex partitioning of the game world geometry. In other words, it is a set of convex polygons that covers the game world, where all adjacent polygons share only two points and one edge, and no polygon overlaps another polygon. Each polygon (or shared edge) represents a waypoint that is connected to the adjacent polygons (see Figure 5.5). Convexity guarantees that we can move in a straight line inside a polygon (e.g. from the current position to the first waypoint, and from the final waypoint to the destination) and from one polygon to another.

By using dynamic programming, we can solve the convex partition problem (i.e. minimize the number of convex polygons needed to cover the original polygon) optimally in the time $O(r^2 n \log n)$, where n is the number of points (i.e. vertices) and r is the

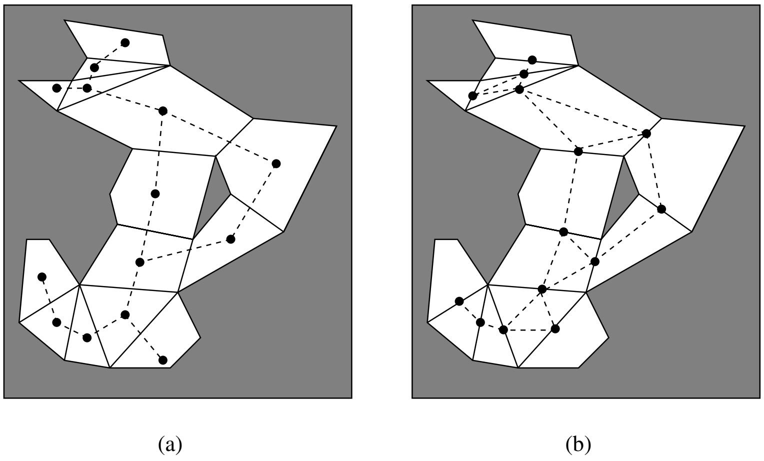

Figure 5.5 Navigation mesh is a convex partitioning of the game world geometry. (a) The waypoints have been placed in the middle of each polygon. (b) The centre of each shared edge is a waypoint.

Algorithm 5.1 Hertel–Mehlhorn method for convex partition.

```
Convex-Partition($P$)
  in:  polygon $P$
  out: convex partition $R$
  1: $R \leftarrow$ Triangulate($P$)
  2: for all $e \in E(R) \setminus E(P)$ do        ▷ Edges added by triangulation.
  3:   if not $e$ divides a concave angle in $R$ then
  4:     $E(R) \leftarrow E(R) \setminus \{e\}$
  5:   end if
  6: end for
  7: return $R$
```

number of notches (i.e. points whose interior angle is concave; $r \leq n - 3$) (Keil 1985). Hertel–Mehlhorn heuristic finds a convex partition in the time $O(n + r \log r)$, and the resulting partition has at most four times the number of polygons of the optimum solution (Hertel and Mehlhorn 1985). The method, described in Algorithm 5.1, first triangulates the original polygon. Although a simple polygon can be triangulated in the time $O(n)$ (Chazelle 1991), Seidel's algorithm provides a simpler randomized algorithm with expected running time of $O(n \log^* n)$ (Seidel 1991). After triangulation, the Hertel–Mehlhorn removes non-essential edges between convex polygons (see Figure 5.6).

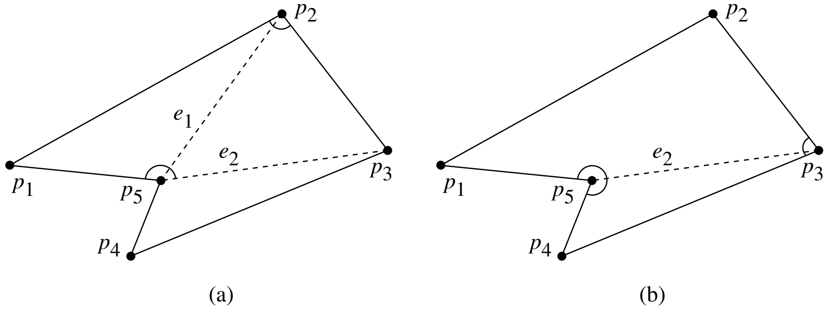

Figure 5.6 After triangulation, Hertel–Mehlhorn method begins to remove non-essential edges to form a convex partition. (a) Edge e_1 is non-essential, because it divides only convex angles at points p_2 and p_5. If it is removed, the resulting polygon $\langle p_1, p_2, p_3, p_5\rangle$ will be convex. (b) When e_1 is removed, edge e_2 becomes essential and cannot be removed, because it divides a concave angle at point p_5.

5.2 Finding the Minimum Path

After the game world have been discretized, the problem of path finding has been transposed into that of path finding in a finite graph. The waypoints are the vertices of the graph, the connections are the edges, and if each connection is associated with a cost (e.g. travelling time), this is assigned to the weight of the edge.

We have a set of well-known graph algorithms for solving the shortest path problem (let $|V|$ be the number of vertices and $|E|$ the number of edges); for details, see Cormen *et al.* (2001).

- *Breadth-first search*: Expand all vertices at distance k from the start vertex before proceeding to any vertices at distance $k + 1$. Once this frontier has reached the goal vertex, the shortest path has been found. The running time is $O(|V| + |E|)$.

- *Depth-first search*: Expand an undiscovered vertex in the neighbourhood of the most recently expanded vertex, until the goal vertex has been found. The running time is $\Theta(|V| + |E|)$.

- *Dijkstra's algorithm*: Find the shortest paths from a single start vertex to all other vertices in a directed graph with non-negative weights. A straightforward implementation yields a running time of $O(|V|^2)$, which can be improved to $O(|V| \log |V| + |E|)$ with a proper choice of data structure.

We can improve the methods by guiding the search heuristically so that as few vertices as possible are expanded during the search. For instance, *best-first search* orders the vertices in the neighbourhood of a vertex according to a heuristic estimate of their closeness to the goal. Despite the use of a heuristic, best-first search returns the optimal solution because no vertex is discarded. Naturally, we can decrease the running time if we give up optimality: *Beam search* is based on best-first search but it expands only the most promising candidates in the neighbourhood thus allowing suboptimal solutions (see Figure 5.7).

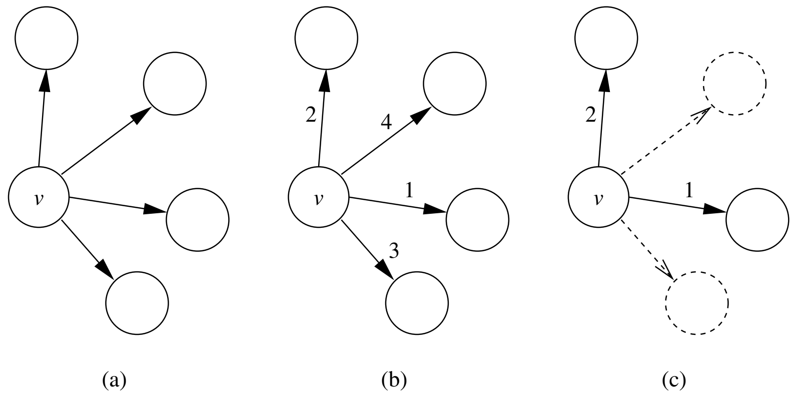

Figure 5.7 Expanding the vertices in the neighbourhood. (a) Breadth-first search does not consider in which order the neighbourhood of vertex v is expanded. (b) Best-first search uses a heuristic function to rank the neighbours but does not exclude any of them. (c) Beam search expands only a subset of the neighbourhood.

In the remainder of this section we consider the properties of a heuristic evaluation function used in guiding the search. Also, we describe and analyse A* algorithm, which is the *de facto* method for path finding in commercial computer games. The basic graph notations used in this section are introduced in Appendix A. The cost of moving (i.e. the sum of weights along a path) from vertex v to vertex u is stored in $g(v \rightsquigarrow u)$. Also, we need two distinguished vertices: a start vertex $s \in V$ and a goal vertex $r \in V$. Obviously, we are interested in the cases in which $s \neq r$, and we want to find a path minimizing $g(s \rightsquigarrow r)$.

5.2.1 Evaluation function

The vertex chosen for expansion is always the one minimizing the evaluation function

$$f(v) = g(s \rightsquigarrow v) + h(v \rightsquigarrow r), \tag{5.1}$$

where $g(s \rightsquigarrow v)$ estimates the minimum cost from the start vertex s to vertex v, and $h(v \rightsquigarrow r)$ is a heuristic estimate of the cost from v to the goal vertex r. Hence, $f(v)$ estimates the minimal cost of the path from the start vertex to the goal vertex passing through vertex v.

Let $g^*(s \rightsquigarrow v)$ denote the exact cost of the shortest path from s to v, and $h^*(v \rightsquigarrow r)$ denote the exact cost of the shortest path from v to r. Now, $f^*(v) = g^*(s \rightsquigarrow v) + h^*(v \rightsquigarrow r)$ gives the exact cost of the optimal path from s to r through vertex v. Ideally, we would use the function f^* in our algorithm, because then we would not have to expand any unnecessary vertices. Unfortunately, for most search problems, such an oracle function h^* does not exist or it is too costly to compute.

The value of the cost function $g(s \rightsquigarrow v)$ is calculated as the actual cost from the start vertex s to vertex v along the cheapest path found so far. If the graph G is a tree, $g(s \rightsquigarrow v)$ will give the exact cost, because there is only one path leading from s to v. In general

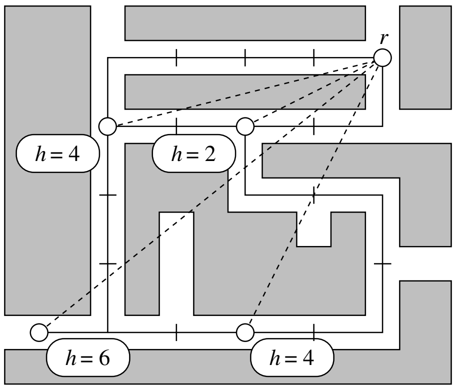

Figure 5.8 An example of a heuristic function. If the weight of an edge is the distance between the vertices using Manhattan metric, a heuristic function h can estimate them with truncated Euclidean distances.

graphs, the cost function $g(s \rightsquigarrow v)$ can err only in overestimating the minimal cost, and its value can be adjusted downwards if a cheaper path to v is found. If we let evaluation function $f(v) = g(s \rightsquigarrow v)$ and assume a cost of 1 unit for each move, we get breadth-first search, because shorter paths will be preferred over the longer ones; instead, if we let $f(v) = -g(s \rightsquigarrow v)$, we get depth-first search, since vertices deeper in the graph will now have a lower cost.

The heuristic function h carries information that is usually based on knowledge from outside the graph. It can be defined in any way appropriate to the problem domain (see Figure 5.8). Obviously, the closer the heuristic estimate is to the actual cost, the lesser our algorithm will expand the superfluous vertices. If we disregard h and our search is based solely on the value of g, we have cheapest-first search, where the algorithm will always choose the vertex nearest to the start vertex. Conversely, an algorithm using only the function h gives us the best-first search.

5.2.2 Properties

Let us define Algorithm A – a name due to tradition – as a best-first search using the evaluation function of Equation (5.1). A search algorithm is *admissible* if it is guaranteed to find a solution path of minimal cost if any solution path exists (e.g. breadth-first search is admissible). If Algorithm A uses the optimal evaluation function f^*, we can prove that it is admissible. In reality, however, the heuristic function h is an estimate. Let us define Algorithm A* as Algorithm A which uses such an estimate. It can be proven that Algorithm A* is admissible if it satisfies the following condition: The value of $h(v \rightsquigarrow r)$ must not overestimate the cost of getting from vertex v to the goal vertex r. In other words,

$$\forall v \in V : h(v \rightsquigarrow r) \leq h^*(v \rightsquigarrow r). \tag{5.2}$$

If the heuristic is locally admissible, it is said to be *monotonic*. In this case, when the search moves through the graph, the evaluation function f will never decrease, since the actual cost is not less than the heuristic cost. Obviously, any monotonic heuristic is also admissible. If Algorithm A* is monotonic, it finds the shortest path to any vertex the first time it is expanded. In other words, if the search rediscovers a vertex, we know that the new path will not be shorter than the one found previously. This allows us to make a significant simplification on the implementation of the algorithm, because we can omit the closed list employed by general search strategies.

Let us state an *optimality* result for Algorithm A*:

Theorem 5.2.1 *The first path from start vertex s to goal vertex r found by monotonic Algorithm A* is optimal.*

Proof. We use a proof by contradiction. Suppose we have an undiscovered vertex v for which $f(v) < g(s \rightsquigarrow r)$. Let u be a vertex lying along the shortest path from s to v. Owing to admissibility, we have $f(u) \leq f(v)$, and because u also must be undiscovered, $f(r) \leq f(u)$. In other words, $f(r) \leq f(u) \leq f(v)$. Because r is the goal vertex, we have $h(r \rightsquigarrow r) = 0$ and $f(r) = g(s \rightsquigarrow r)$. From this it follows that $g(s \rightsquigarrow r) \leq f(v)$, which is a contradiction. This means that there does not exist any undiscovered vertices that are closer to the start vertex s than the goal vertex r. ■

Although h is sufficient to be a lower estimate on h^*, the more closely it approximates h^*, the better the search algorithm will perform. We can now compare two A* algorithms with respect to their *informedness*. Algorithm $\mathcal{A}_1$ using function h_1 is said to be more informed than algorithm $\mathcal{A}_2$ using function h_2 if

$$\forall\, v \in V \setminus \{r\} : h_1(v \rightsquigarrow r) \geq h_2(v \rightsquigarrow r). \tag{5.3}$$

This means that $\mathcal{A}_1$ will never expand more vertices than what are expanded by $\mathcal{A}_2$. Because of informedness, there is no better approach than Algorithm A* in the sense that no other search strategy with access to *the same amount of outside knowledge* can do any less work than A* and still be sure of finding the optimal solution.

5.2.3 Algorithm A*

Algorithm 5.2 describes an implementation of Algorithm A*. As mentioned earlier, monotonicity of the evaluation function means that we need only to update an open list of the candidate vertices (lines 15–20), and the algorithm can terminate when it has found the goal vertex (lines 10–12).

Figure 5.9 gives an example of how Algorithm A* works: (a) The weight of an edge describes the distance between its endpoints in Manhattan metric. (b) First, start vertex s gets selected, and its successors a and b are added to the set S (i.e. they are opened). The heuristic measure takes the maximum of the vertical and horizontal distance to the goal vertex. Because $f(b) < f(a)$, vertex b gets selected next. (c) Algorithm opens the successors of vertex b. Vertex e is the most promising candidate and gets selected. (d) Vertex e does not have undiscovered successors, because d was already opened by b. The remaining candidates have the same value, so algorithm selects c arbitrarily. Vertex f is opened. (e) Vertex a has the lowest value but has no undiscovered successors. Instead, vertex d gets selected and g is opened. (f) Of two remaining candidates, vertex g gets selected, and goal vertex r is found. The optimum path is $s \to b \to d \to g \to r$ and its cost is 12.

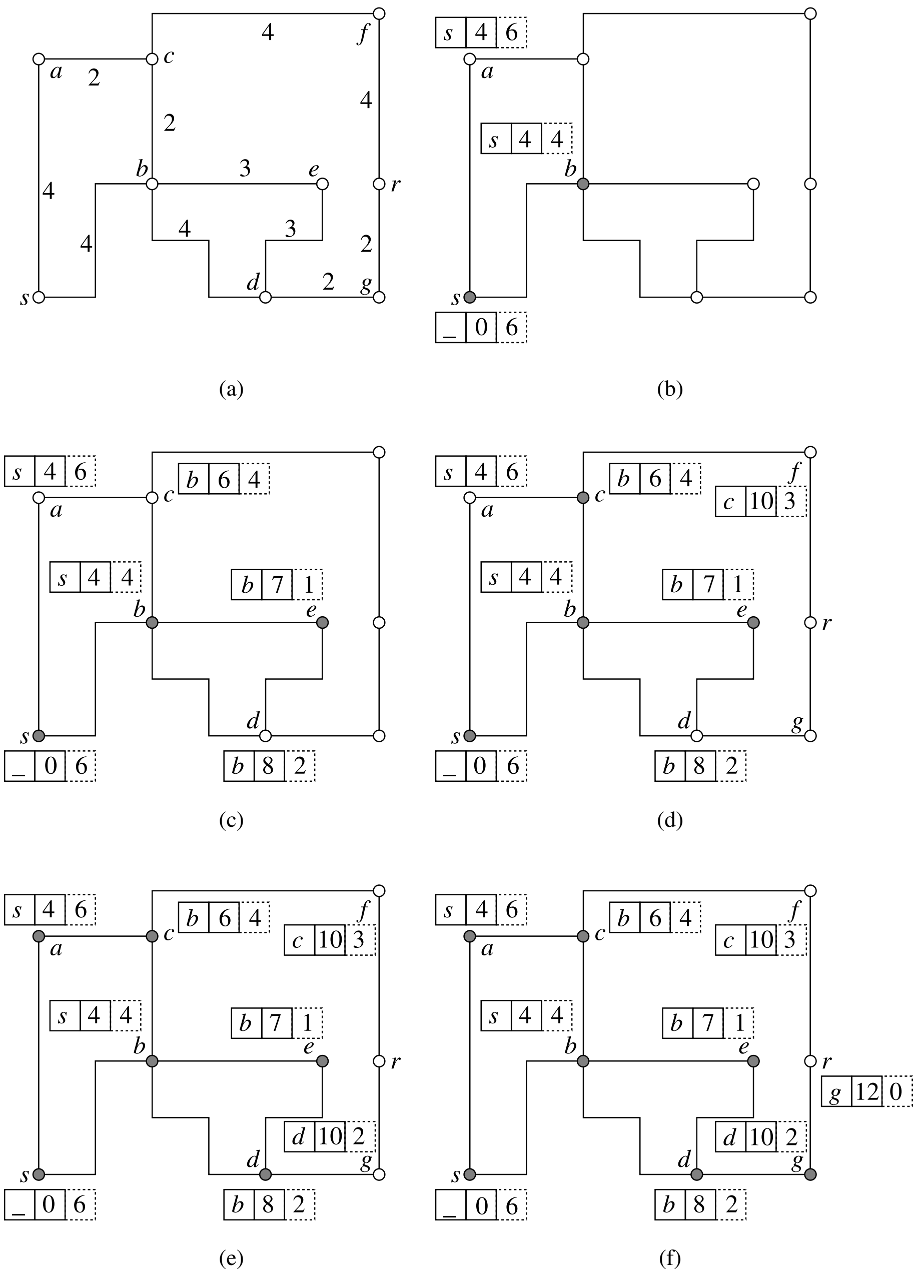

Figure 5.9 An example of Algorithm A*. The boxes next to a vertex v represent values $\pi(v)$, $g(s \rightsquigarrow v)$, and $h(v \rightsquigarrow r)$. Filled circles indicate the selected vertices.

Algorithm 5.2 Algorithm A* for a monotonic evaluation function.

```
A-STAR($G, s, r$)
  in:    graph $G = (V, E)$; start vertex $s$; goal vertex $r$
  out:   mapping $\pi : V \to V$
  local: open list $S$; cost function $g(u \rightsquigarrow v)$; heuristic lower bound estimate $h(u \rightsquigarrow v)$
 1: for all $v \in V$ do                         ▷ Initialization.
 2:   $g(s \rightsquigarrow v) \leftarrow \infty$
 3:   $\pi(v) \leftarrow$ NIL
 4: end for
 5: $g(s \rightsquigarrow s) \leftarrow 0$
 6: $S \leftarrow \{s\}$
 7: precalculate $h(s \rightsquigarrow r)$
 8: while $S \neq \emptyset$ do                   ▷ Search.
 9:   $v \leftarrow$ vertex $v' \in S$ that minimizes $g(s \rightsquigarrow v) + h(v \rightsquigarrow r)$
10:   if $v = r$ then                           ▷ Is the goal reached?
11:     return $\pi$
12:   end if
13:   $S \leftarrow S \setminus \{v\}$
14:   for all $u \in successors(v)$ do
15:     if $\pi(u) =$ NIL or else ($u \in S$ and
          $g(s \rightsquigarrow v) + weight(v, u) < g(s \rightsquigarrow u)$) then          ▷ Open $u$.
16:       $S \leftarrow S \cup \{u\}$
17:       $g(s \rightsquigarrow u) \leftarrow g(s \rightsquigarrow v) + weight(v, u)$
18:       $\pi(u) \leftarrow v$
19:       precalculate $h(u \rightsquigarrow r)$
20:     end if
21:   end for
22: end while
23: error no path from $s$ to $r$ exists
```

Apart from optimality, there may be practical considerations when implementing A*. First, the computational effort depends on the difficulty of computing the function h. If we use less informed – and computationally less intensive – heuristic, we may go through more vertices but, at the same time, the total computation requirement can be smaller. Second, we may content ourselves in finding a solution reasonably close to the optimum. In such a case, we can use a function that evaluates accurately in most cases but sometimes overestimates the cost to the goal, thus yielding an inadmissible algorithm. Third, we can weight (or even change) the heuristic function when the search has proceeded far from the source vertex s. For example, we can use a more precise heuristic for the nearby vertices and approximate the magnitude for the faraway ones. For dynamic graphs (i.e. the waypoints and their relationships can change in the game world), this can even be the best approach, because it is likely that we have to search new path after a while. To summarize, the choice of the function h and the resulting heuristic power of Algorithm A* depend on a compromise among these practical considerations.

5.3 Realizing the Movement

After the path has been solved in a graph, it must be realized in the game world. Although the solution may be optimal in the graph, it can be unrealistic or aesthetically displeasing in the game world (Patel 2003). For example, consider the situation illustrated in Figure 5.10, where a game character has to move from one room to another. Because the character goes through the waypoints, the resulting path has sharp turns instead of a smooth movement. This stems from the selection of the waypoints with respect to the intended movement: The more artificial or 'virtual' the waypoint is, the more unrealistic the movement through it looks. Of course, sharp turns at the wall extensions and beside the door frames can be realistic if the game character is under fire.

Naturally, we can use Bézier curves or B-splines (Watt 2000) instead of following the path in straight lines, but there are simpler approaches. One possibility is to use line-of-sight testing to reduce the number of waypoints the character has to visit (Snook 2000). Figure 5.11 illustrates the situation: Instead of heading to the next waypoint in the path, the character chooses the farthest waypoint it can see and heads there. This is repeated until the destination is reached. The path followed can be further reduced by changing the route to be always towards the farthest visible waypoint.

To avoid (possibly dynamic) obstacles, we can use the avoidance rule of flocking algorithm (see Section 6.3), and assign a repulsion vector to the obstacles (Johnson 2003). Figure 5.12 illustrates a situation in which an obstacle is blocking the direct path. To avoid it the character's velocity vector combines two components, the desired direction towards the destination and the repulsion away from the obstacle, which is enough to steer the character past the obstacle. In other words, force vectors (and vector fields) are a convenient balancing mechanism between local actualizations (i.e. reactive behaviour in the continuous world) and global intentions (i.e. planning in the discretization of the world).

Because path finding can be a time-consuming task, special care must be taken when it is accessed through a user interface. When players give orders to a game character, they expect it to respond immediately, even if the path finding required to comply with the order is not yet finished. One solution is to get the character moving towards the general direction

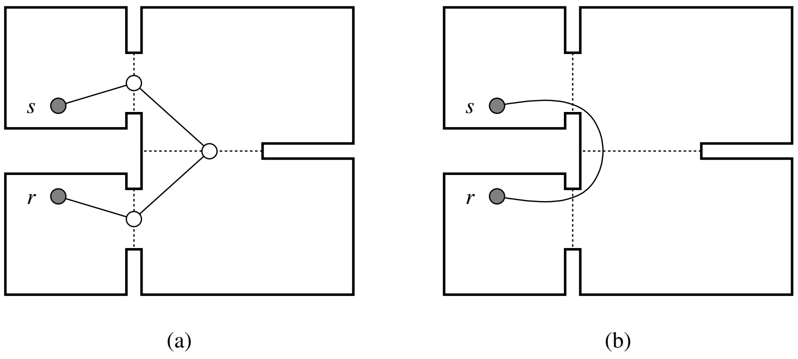

Figure 5.10 (a) The path through the waypoints can have sharp and unrealistic turns. (b) Sharp turns can be smoothed out when the movement is realized.

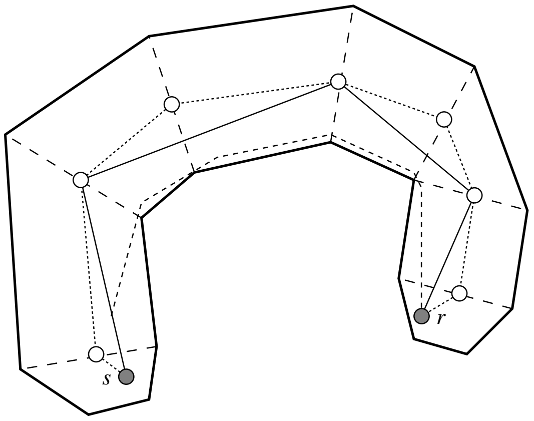

Figure 5.11 Line-of-sight testing allows to improve the original path (dotted line) by skipping waypoints (solid line). It is not necessary to visit the waypoints, but the heading can be changed immediately whenever a farther waypoint becomes visible (dashed line).

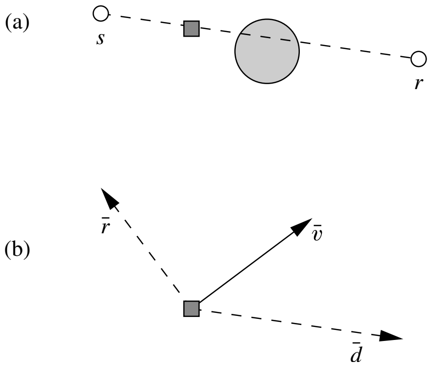

Figure 5.12 Avoiding dynamic obstacles. (a) The straight path from s to r is obstructed. (b) The desired direction $\bar{d}$ towards the destination and the repulsion direction $\bar{r}$ away the obstacle are combined to form the velocity vector $\bar{v}$.

of the destination (or animate that it is preparing to move), while the full path is still being calculated (Higgins 2002). When the path finding is ready, the character, which has moved somewhat, is redirected to the found path.

5.4 Summary

The future path for path finding is still unsolved. Many alternative methods have been proposed, but the three-stage approach presented in this chapter is still the standard approach in commercial computer games. Its main advantage is that we decompose the problem into more manageable sub-problems, each of which has set readily available, reliable, and reasonably fast solution methods.

Reactive agents from robotics have been proposed for solving the path-finding problem. They reduce the solution method to simple reactive rules akin to the flocking algorithm, and the emerging behaviour finds a path for the agent. At the moment the intelligence of these methods is at the level of insects, and, no matter however intelligent insects can be, designing a usable method for computer games seems a difficult task.

Analytical approaches take the opposite view and say that the more data we have the better. They try to solve path finding straightforwardly by modelling all related factors – which may sound good in theory, but in practice some relevant details can escape precise mathematical formulation.

A third approach suggested to solve path finding is artificial intelligence (AI) processors. The idea is that the usual methods for solving AI problems – including path finding – can be made into a hardware component much like 3D display cards, which would take away many time-consuming tasks from the software. Unfortunately, at the time of writing this book, this seems to be still some time ahead in the future. Also, the method used in the AI processor has to be based on some existing software solution – possibly the one presented here.

Exercises

5-1 Imagine that you would have to describe a route to a blindfolded person and describe how to get

(a) from the kitchen to the living room.

(b) from home to work/school.

(c) from home to Rome.

Be as accurate as necessary in your description.

5-2 A monkey is in a cage formed by a square grid (see Figure 5.13). He is hungry but cannot reach the banana dangling from the ceiling. There is a box inside the cage,

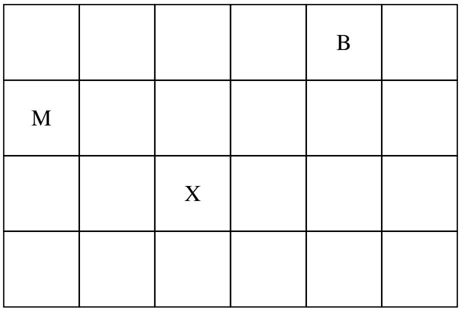

Figure 5.13 Monkey (M), box (X) and banana (B) in a cage formed by a square grid.

and the monkey can reach the banana if the box is underneath the banana. If the monkey is beside the box, he can lift it to one of the neighbouring tiles. The problem is to find a sequence of moves through which the monkey can get the banana from any given initial situation. The monkey sees the whole situation and can select the following operations: move to a tile in the grid, lift the box to a neighbouring tile, get on the box, get down from the box, and reach for the banana.

Form this monkey-in-a-cage problem as a path-finding problem and design an algorithm to solve it.

5-3 Waypoints can be laid down manually by the game world designer. What benefits and drawbacks does this have over the automated waypoint assigning process?

5-4 Prove that there are only three regular two-dimensional edge-sharing tessellations.

5-5 To have random-access lookup, a grid should have a scheme for numbering the tiles. For example, a square grid has rows and columns, which give a natural numbering for the tiles. Devise schemes for triangular and hexagonal grids. Use the numbering scheme to define a rule for determining the neighbourhood (i.e. adjacent tiles) of a given tile in the grid. For example, if we have a four-connected square grid, where the indices are i for rows and j for columns, the neighbourhood of tile $\langle i, j\rangle$ can be defined as

$$\mathit{neighbourhood}(\langle i, j\rangle) = \{\langle i \pm 1, j\rangle, \langle i, j \pm 1\rangle\}.$$

5-6 A hexagonal grid is not straightforward enough to be represented on the screen (i.e. using square pixels). Devise an algorithm for displaying it.

5-7 Let us connect Exercise 5-5 and Exercise 5-6 and define a mapping τ from a position in a continuous game world to its corresponding tile number. For example, if we are using a square grid with edge length ℓ, we can define $\tau : \mathbb{R}^2 \to \langle \mathbb{N}, \mathbb{N}\rangle$ straightforwardly as

$$\tau : (x, y) \mapsto \langle \lfloor x/\ell \rfloor, \lfloor y/\ell \rfloor\rangle.$$

Write algorithms that calculate τ for triangular and hexagonal grids.

5-8 Triangulate the game world of Figure 5.14. Then apply the Hertel–Mehlhorn method and remove excess edges.

5-9 For what kind of search problems does breadth-first and depth-first suit best?

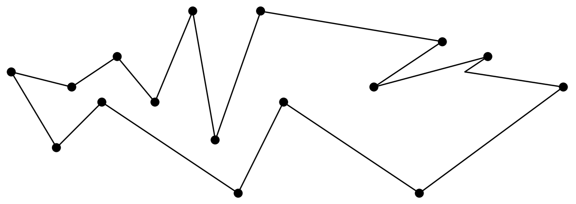

Figure 5.14 A game world as a polygon.

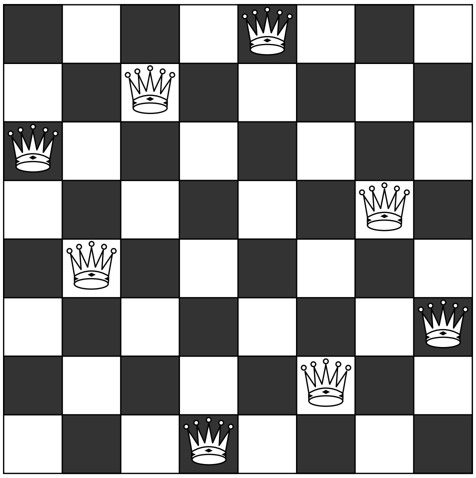
Figure 5.15 One possible solution to the 8 queens problem.

5-10 In n queens problem, n queens should be placed on an $n \times n$ chessboard so that they do not threaten each other. Figure 5.15 gives one solution to the 8 queens problem, which has in total – omitting rotations and mirror images – 12 different solutions. Formulate the n queens problem as a search problem.

5-11 If we had an oracle function $h^*(v \rightsquigarrow r)$, which gives the exact cost of getting from v to r, how could we solve the minimum path search problem? Why is such a function so hard to form?

5-12 Although A* algorithm works better with a more informed heuristic function, the overall computing time can be smaller with a less informed heuristic. How can that be possible?

5-13 What happens to the paths if we use an inadmissible search algorithm?

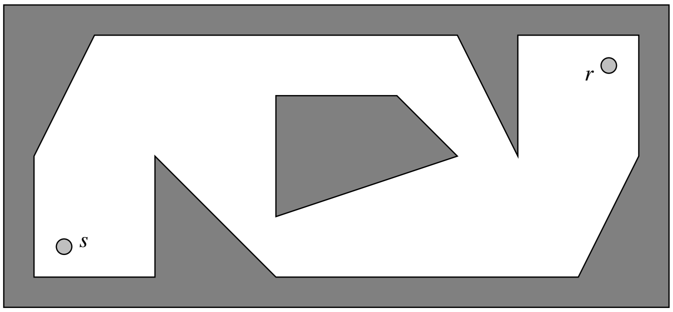

Figure 5.16 Two-dimensional game world, where white area represents open space.

5-14 What other aesthetic considerations are there in movement realization besides smooth movements and obstacle avoidance?

5-15 Assume we have the game world of Figure 5.16. The player wants to move from the point s to the point r using only the white area (i.e. the path cannot go to the grey area). How would you solve this path-finding problem? Describe the three phases of the approach in general terms. Select a method for each phase and apply them to the given problem instance to find a path from s to r.