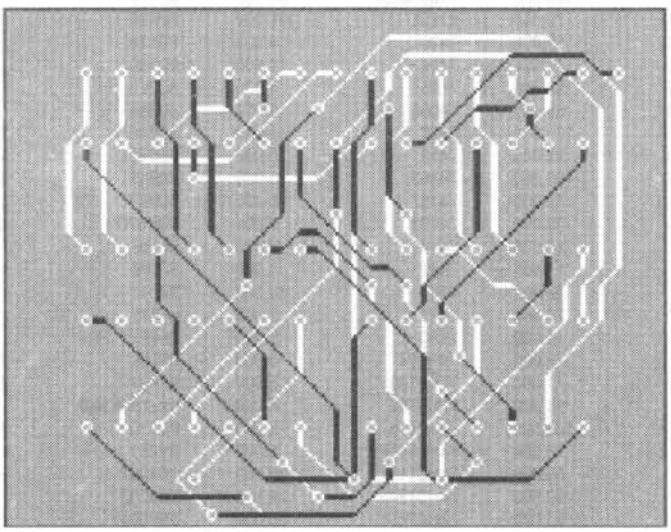
Zestaw programów Randy Nevin'a

Automatyczne projektowanie mozaiki ścieżek metodą breadth-first i A*

W trakcie pracy z programami PCB etap tworzenia ścieżek na podstawie listy połączeń może zostać wykonany automatycznie. Często zdarza się, że obserwując na ekranie przebieg procesu projektowania pomrukujemy niezadowoleni pod nosem, dziwiąc się wybieranym przez program rozwiązaniom. Nie zdajemy sobie zazwyczaj sprawy, że skonstruowanie efektywnego algorytmu, generującego globalnie optymalne połączenia, jest zadaniem niezwyke skomplikowanym (o ile w ogóle możliwym). Jeśli ktoś zna język C i chce się o tym przekonać, to ma niepowtarzalną okazję spróbować własnych sił... Dobrym pretekstem do poruszenia tej problematyki na łamach EP jest prezentacja kolejnej pozycji z oferty programów shareware.



Rys. 1. Wykonany projekt płytki

Problem automatycznego projektowania ścieżek może być potraktowany jako globalny problem optymalizacji. Realizowany układ można scharakteryzować przy pomocy wielu zmiennych, takich jak: długość ścieżek, wymiary płytki, liczba przelotek, współczynniki sprzężeń pasożytniczych, ilość warstw, koszt wykonania, przejrzystość połączeń, podatność na usuwanie błędów itd.. Aby dobrze wykonać projekt, dążymy do minimalizacji niektórych z nich (np. długość ścieżek), zaś niektóre chcemy zmaksymalizować (np. podatność na usuwanie błędów). Jakość wykonanego układu jest funkcją tych często kolidujących ze sobą zmiennych. Jako zadowalające przyjmuje się rozwiązanie, w którym układ jedynie wykazuje się założonymi parametrami, ponieważ znalezienie globalnie optymalnego rozwiązania jest nieosiągalne (z wyjątkiem przypadków trywialnych).

Automatyczne projektowanie ścieżek można również potraktować jako grupę problemów odnajdywania drogi między dwoma punktami na płytce, którą dla potrzeb algorytmów modeluje się w postaci matrycy komórek. Istnieje wiele dających się tu zastosować algorytmów, różniących się możliwościami i parametrami użytkowymi. Zostaną przedstawione dwa z nich, określane w literaturze jako *breadth-first* i *A**.

Algorytmy odnajdywania drogi działają zwykle dwufazowo. Faza pierwsza rozpoczyna się w jednej z komórek, które chcemy połączyć, zwanej komórką źródłową (lub źródłem) - zadaniem jest dotarcie do drugiej komórki, zwanej komórką docelową (lub celem). Poszukiwania są zazwyczaj prowadzone w wielu kierunkach jednocześnie. W trakcie poszukiwań są zapamiętywane informacje o tym, jak każda z komórek została znaleziona. Odnalezienie komórki docelowej kończy pierwszą fazę poszukiwań.

Rozpoczyna się faza druga, w której na podstawie zapamiętanych w fazie pierwszej informacji określa się drogę, po której komórka docelowa została osiągnięta i wykonuje się połączenie. Jeśli faza pierwsza wyczerpała już wszystkie możliwości poszukiwań, a cel nie został osiągnięty, wówczas oznacza to, że nie istnieje możliwość realizacji połączenia między komórkami.

Druga faza jest identyczna dla obu wspomnianych algorytmów. Istotne różnice występują w fazie pierwszej i to powoduje, że algorytmy zachowują się inaczej.

Dla potrzeb poszukiwań organizowane są dwie struktury danych przechowujące współrzędne komórek (ponieważ współrzędne w sposób jednoznaczny identyfikują komórki, można powiedzieć po prostu, że struktury te zawierają komórki):

- kolejka o nazwie OTWARTE zawierająca komórki odnalezione, lecz oczekujące na przetworzenie przez algorytm poszukiwań,
- zbiór o nazwie ZAMKNIĘTE przechowujący komórki odnalezione i przetworzone.

Komórki nie znajdujące się w kolejce OTWARTE ani w zbiorze ZAMKNIĘTE będą określane jako wolne. Komórki bezpośrednio do siebie przylegające będą nazywane sąsiadami.

Na początku poszukiwań kolejka OTWARTE zawiera jedynie komórkę źródłową, a zbiór ZAMKNIĘTE jest pusty. Faza pierwsza jest realizowana przy pomocy pętli, w której pobierana jest komórka z kolejki OTWARTE, następnie jest ona wstawiana do zbioru ZAMKNIĘTE, po czym natępuje sprawdzenie, czy jest to komórka docelowa. Jeśli tak, to faza pierwsza jest zakończona. Jeżeli nie, to do kolejki OTWARTE trafiają sąsiedzi komórki i pętla powtarza się. Jak za chwilę zobaczymy, główną różnicą między algorytmami *breadth-first* i *A** jest porządek, w jakim sąsiedzi komórki trafiają do kolejki.

Breadth-first

Formalny zapis algorytmu *breadth-first* znajduje się na **listingu 1**. Algorytm pracuje z wykorzysta-

```
BFS Algorithm  (* breadth-first search *)
   (* Search a graph or state space, depending on the problem definition. *)
   (* S is the start node, T is the goal node. *)
   (* Open is an ordered list of nodes (ordered by arrival time; nodes enter
      at the tail and leave at the head), also called a queue. Closed is a set
      of nodes (order doesn't matter). In general, nodes that need to be
      searched are put on Open. As they are searched, they are removed from
      Open and put in Closed. *)
   (* Pred is defined for each node, and is a list of "came from" indications,
      so when we finally reach T, we traverse Pred to construct a path to S. *)
1  Open <- {S}  (* a list of one element *)
   Closed <- {}  (* the empty set *)
   Pred[S] <- NULL,  found <- FALSE
   WHILE Open <> {} and not found DO
5     x <- the first node on Open
      Open <- Open - {x}  (* remove x from Open *)
      Closed <- Closed + {x}  (* put x in Closed *)
      IF x = T THEN found <- TRUE  (* we're done *)
      ELSE  (* continue search through node x *)
10       let R be the set of neighboring nodes of x
         FOR each y in R DO
            IF y is not on Open or in Closed THEN
               Pred[y] <- x  (* remember where we came from *)
               Open <- Open + {y}  (* put y on Open (at the tail) *)
15 IF found THEN
      use Pred[T] to find Pred[Pred[T]] and so on until S is reached
      (* this traces out the solution path in reverse *)
   ELSE T cannot be reached from S
```

List. 1. Algorytm *breadth-first*

niem klasycznej kolejki, czyli struktury FIFO (od angielskiego określenia First In First Out; jest to zwięzłe wyrażenie właściwości kolejki, która charakteryzuje się tym, że zwraca najwcześniej wstawiony do niej element).

Wersja *breadth-first*, znana pod nazwą algorytmu Leego, została zastosowana do automatycznego projektowania połączeń już we wczesnych latach '60 i ciągle jest podstawą działania wielu programów PCB. W oryginalnej postaci algorytmu komórki przylegające do siebie diagonalnie nie były uznawane za sąsiadów, dlatego faza druga nie mogła wykonywać połączeń „na skos".

W algorytmie *breadth-first* do kolejki trafiają tylko wolni sąsiedzi, a kolejność, w której są wstawiani nie jest określona.

Algorytm ten ma poważną wadę: jeśli dystans między źródłem a celem wzrośnie N razy, to liczba komórek przetwarzanych w czasie poszukiwań (a tym samym czas poszukiwań) zwiększa się z kwadratem N. Można to łatwo uzmysłowić, porównując proces poszukiwań w algorytmie *breadth-first* do rozprzestrzeniania się kolistych fal po wrzuceniu kamienia do stawu. Jeżeli kamień rzucimy N razy dalej (oddalimy punkt źródłowy), to pofalowana powierzchnia stawu (obszar poszukiwań), zanim fale dotrą do nas (cel), będzie równa (co wynika z podstawowych praw geometrii):

$PI * N^2 * r^2$

gdzie r oznacza odległość pierwszego rzutu. Zatem obszar poszukiwań wzrośnie

$(PI * N^2 * r^2)/(PI * r^2) = N^2$

razy.

Wykonanie algorytmu *breadth-first* może więc zająć dużo czasu. Niewątpliwą jego zaletą jest prostota, nie zmienia to jednak faktu, że w praktyce można go stosować do rozwiązywania jedynie niewielkich problemów.

A*

Formalny zapis algorytmu *A** znajduje się na **listingu 2**.

W algorytmie tym kolejka OTWARTE występuje w postaci zmodyfikowanej: w zależności od pewnej właściwości komórki, może ona być wstawiana na miejsca inne niż koniec (co jest niemożliwe do wykonania w klasycznej FIFO stosowanej w *breadth-first*). Struktura taka jest nazywana kolejką z priorytetem.

Czynnikiem określającym priorytet wstawianej komórki x jest w wypadku algorytmu *A** przybliżona odległość celu H(x). Im jest ona krótsza, tym bliżej początku kolejki może znaleźć się komórka.

W pętli iteracyjnej z kolejki jest pobierana komórka, która jest wstawiana do zbioru ZAMKNIĘTE, a jej wolni sąsiedzi trafiają na odpowiednie miejsca w kolejce. Każdy sąsiad x, który znalazł się już w zbiorze ZAMKNIĘTE, jest sprawdzany, czy nowa droga D(x) od źródła do niego jest lepsza (krótsza) od poprzedniej. Jeśli warunek ten jest spełniony, to wraca na odpowiednie miejsce do kolejki zależne od wartości jego H(x) (zauważmy, że D(x), czyli długość drogi przebytej ze źródła do komórki x, jest znana dokładnie w odróżnieniu od odległości celu H(x), która jest wartością przybliżoną). Wymienione czynności są powtarzane aż do osiągnięcia celu lub opróżnienia kolejki (co jest równoważne z wyczerpaniem się możliwości poszukiwań).

```
A* Algorithm  (* heuristic search *)
   (* Search a graph or state space, depending on the problem definition. *)
   (* S is the start node, T is the goal node. *)
   (* Open is an ordered list of nodes (ordered by lowest F value; see below),
      also called a priority queue. Closed is a set of nodes (order doesn't
      matter). In general, nodes that need to be searched are put on Open (at
      the proper position). As they are searched, they are removed from Open
      and put in Closed. Occasionally a newer, better route will be found to a
      node after it has already been searched, in which case we remove it from
      Closed and put it back on Open to be reconsidered. *)
   (* G[x] is the distance already traveled to get from S to node x, and is
      known exactly. H(x) is a function (heuristic) which returns an estimate
      of the distance from node x to T. F[x] is the estimated distance from S
      to T by going through node x, and is computed by F[x] = G[x] + H(x).
      H(x) can be calculated for any node, but F[x] and G[x] only become
      defined when node x is visited. *)
   (* Pred is defined for each node, and is a list of "came from" indications,
      so when we finally reach T, we traverse Pred to construct a path to
      S. *)
   (* Distance(x,y) is a function for calculating the distance between two
      neighboring nodes. *)
1  Open <- {S}  (* a list of one element *)
   Closed <- {}  (* the empty set *)
   G[S] <- 0,  F[S] <- 0,  Pred[S] <- NULL,  found <- FALSE
   WHILE Open <> {} and not found DO
5     x <- the first node on Open  (* node with smallest F value *)
      Open <- Open - {x}  (* remove x from Open *)
      Closed <- Closed + {x}  (* put x in Closed *)
      IF x = T THEN found <- TRUE  (* we're done *)
      ELSE  (* continue search through node x *)
10       let R be the set of neighboring nodes of x
         FOR each y in R DO
            IF y is not on Open or in Closed THEN
               G[y] <- G[x] + Distance(x,y)
               F[y] <- G[y] + H(y)  (* estimate solution path length *)
15             Pred[y] <- x  (* remember where we came from *)
               Open <- Open + {y}  (* put y on Open *)
            ELSE  (* y is on Open or in Closed *)
               IF (G[x] + Distance(x,y)) < G[y] THEN
                  (* we've found a better route to y *)
20                G[y] <- G[x] + Distance(x,y)
                  F[y] <- G[y] + H(y)
                  Pred[y] <- x  (* remember where we came from *)
                  IF y is on Open THEN
                     reposition y according to F[y]
25                ELSE  (* y is in Closed *)
                     Closed <- Closed - {y}  (* remove y from Closed *)
                     Open <- Open + {y}  (* put y on Open *)
   IF found THEN
      use Pred[T] to find Pred[Pred[T]] and so on until S is reached
30    (* this traces out the solution path in reverse *)
   ELSE T cannot be reached from S
```

List. 2. Algorytm *A**

Cechą algorytmu jest konieczność wyznaczania przybliżonej wartości odległości do celu H(x). Od dobrej estymacji tej wartości zależy możliwość koncentracji działań algorytmu w kierunku dającym największe prawdopodobieństwo zakończenia poszukiwań sukcesem. Im zastosowana metoda przybliżająca jest lepsza, tym szybsze działanie algorytmu. Do celów PCB można z powodzeniem stosować proste aproksymacje geometryczne.

W praktyce okazuje się, że algorytm *A** nie cierpi na „kwadratowy rozrost" metody *breadth-first* i w związku z tym rozwiązuje te same problemy szybciej. Niestety, ze względu na konieczność przechowywania warości D(x) każdej odszukanej komórki, metoda *A** wymaga większych ilości pamięci.

Doświadczenia pokazują, że jeśli zwiększyć dwa razy odległość między źródłem i celem, to liczba przetworzonych w czasie poszukiwań komórek również wzrośnie dwa razy, po potrojeniu odległości - trzy razy, itd. To liniowe zachowanie się algorytmu jest znacznie bardziej atrakcyjne z punktu widzenia jego implementacji w programach PCB niż „kwadratowe" zachowanie się algorytmu *breadth-first*.

Metody stosowane w praktyce są znacznie bardziej rozbudowane od opisanych. Zauważmy, że nie uwzględnialiśmy w rozważaniach połączeń zaprojektowanych wcześniej, płytek wielowarstwowych, czy możliwości tworzenia ścieżki do celu, który jest obiektem innym niż punkt (np. nieregularny obszar wypełniony miedzią stanowiący masę układu). Problem dodatkowo się komplikuje, jeżeli zechcemy uwzględniać w jakiś sposób następstwa kształtu wykonywanego połączenia dla stopnia komplikacji lub możliwości realizacji następnych połączeń. Wszystko to powoduje, że w chwili obecnej wprawne oko i intuicja projektanta są nadal podstawowymi środkami prowadzącymi do ostatecznej optymalizacji projektów, zwłaszcza tych bardziej rozbudowanych.

Nowego (być może nawet rewolucyjnego) podejścia do problemu może dostarczyć nadchodząca era neurokomputerów i oprogramowania genetycznego (postaramy się o tym napisać w niedalekiej przyszłości).

Osoby zainteresowane tematyką algorytmów projektowania ścieżek zachęcamy do zapoznania się z zestawem PCB autorstwa Randy Nevin'a. Znajdują się w nim:

- pliki *.C, *.ASM zawierające kody źródłowe programów, które realizują opisane algorytmy; napisane zostały dla kompilatora Microsoft C 4.0 oraz assemblera Microsoft MASM 4.0, ale mogą być łatwo zaadaptowane dla potrzeb innych kompilatorów/assemblerów,
- programy wykonywalne *.EXE będące wynikiem kompilacji kodów źródłowych,
- pliki *.INC zawierające definicje podstawowych układów TTL,
- pliki zawierające dokumentację (między innymi bardziej szczegółowe informacje na temat metod *A** i *breadth-first*).

Wszystkie programy zestawu wymagają argumentów przekazywanych w linii komend; programy uruchomione bez nich wyświetlają jedynie informację o tym, jakie argumenty są potrzebne i w jakiej kolejności. Dla PCBROUTE.EXE potrzebne są dwa argumenty: nazwa pliku zawierającego opis połączeń układu oraz nazwa pliku, w jakim ma zostać zapisany wykonany projekt płytki. Dla PCBPRINT.EXE i PCBVIEW.EXE wymagany jest jeden argument - nazwa pliku zawierającego projekt płytki układu. Program PCBPRINT.EXE ma cztery opcjonalne argumenty, które umożliwiają:

- wybór rozdzielczości drukarki (75, 100, 300 dpi),
- powiększenie wydruku (maksymalnie 3 razy),
- wydruk typu „landscape" ("bokiem"),
- wydruk typu „portrait".

Program PCBROUTE.EXE odczytuje plik wejściowy zawierający opis połączeń, projektuje ścieżki i zapisuje wynik w pliku wyjściowym. Plik wejściowy jest plikiem tekstowym i może zostać stworzony przy pomocy dowolnego edytora. W zestawie znajduje się plik przykładowy EXAMPLE, na podstawie którego można zaznajomić się z formatem opisu połączeń układu używanym przez program PCBROUTE.EXE. Projekt płytki można obejrzeć używając programu PCBVIEW.EXE (w trybie EGA 640x350) lub wydrukować przy pomocy programu PCBPRINT.EXE (na drukarce Hewlett-Packard LaserJet).

Zestaw posiada wyjątkowe walory dydaktyczne i poznawcze. Polecamy go tym użytkownikom PCB, których nie zadowala czysto konsumpcyjne podejście do oprogramowania, lecz czasami zadają sobie pytania „jak?" i "dlaczego?".

Kody źródłowe dołączone do zestawu są niezwykle atrakcyjnym materiałem dla własnych modyfikacji i doświadczeń; napisano je przejrzyście i bogato opatrzono komentarzami. Warto się przyjrzeć, jak programują najlepsi... (autor zestawu Randy Nevin jest pracownikiem firmy Microsoft).

Zestaw programów jest dostępny w ofercie AVT programów shareware na dyskietce 1CA011.

Opr. S.A.M.