České vysoké učení technické v Praze
Fakulta elektrotechnická

Bakalářská práce

Interaktivní nástroj pro kreslení schémat logických obvodů

Robert Škorpil

Vedoucí práce: Ing. Petr Fišer, Ph.D.

Studijní program: Elektrotechnika a informatika strukturovaný bakalářský

Obor: Informatika a výpočetní technika

červenec 2008

Poděkování

Chtěl bych tímto poděkovat vedoucímu této práce, panu Ing. Petru Fišerovi, Ph.D., za jeho cenné rady.

Prohlášení

Prohlašuji, že jsem svou bakalářskou práci vypracoval samostatně a použil jsem pouze podklady uvedené v přiloženém seznamu.
Nemám závažný důvod proti užití tohoto školního díla ve smyslu §60 zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon).

V Praze dne 10. 7. 2008 .....

Abstract

The goal of this thesis was to integrate automatic wire connecting feature into the existing application INKS (Interactive Tool for Drawing Schematics of Logic Circuits). It examines utilization possibilities of several algoritms and presents the chosen solution, which is modified A* algorithm. It describes the implementation of the algorithm including the integration into the existing program and needed alterations of the program. It evaluates achieved results and compares them to a proprietary application's behaviour.

Abstrakt

Úkolem této práce je doplnění automatického propojování spojů do stávajícího programu pro interaktivní kreslení schémat logických obvodů. Rozebírá možnosti využití některých algoritmů a předkládá vlastní zvolené řešení, kterým je modifikovaný algoritmus A*. Popisuje způsob implementace algoritmu včetně integrace do stávajícího programu a úprav, které v něm bylo nutné provézt. Zhodnocuje dosažené výsledky a porovnává je s komerčním programem.

Obsah

Seznam obrázků

Seznam tabulek

1 Úvod

Úkolem této práce bylo doplnění funkce automatické propojování do stávajícího interaktivního nástroje pro kreslení schémat (INKS). Program INKS je nástroj pro grafický návrh schémat logických obvodů, jehož autorem je Bc. Rostislav Pastor. V současné verzi měl program tyto funkce[1]:

- Kreslení schémat pomocí předdefinových symbolů
- Ukládání a načítání schémat
- Podpora hierarchie schémat, vytváření funkčních bloků
- Postskriptový výstup schémat

Stávající program vyžadoval při vytváření spojů, aby uživatel myší ručně vyznačil dráhu celého spoje. Ve srovnání s obdobnými programy, které toto nevyžadují se ovšem takové chování programu dá považovat za nepříliš komfortní. Absence této funkcionality ve stávajícím programu se zároveň projevovala při úpravách schématu. Při posunu hradla připojeného spoji k ostatním objektům ve schématu nedocházelo k přizpůsobení dráhy spojů. Program také neumožňoval označování více objektů a jejich současný posun. Takové chování ovšem brání programu v možnosti jeho využití jako plnohodnotného nástroje. Mým úkolem bylo tedy nástroj v tomto smyslu učinit přívětivější k uživateli.

2 Popis problému, specifikace cíle

2.1 Popis problému

Funkce automatického propojování v nástroji pro vytváření logických schémat je problém, který se velmi podobá problému z jiné oblasti nástrojů EDA (electronic design automation). Jedná se o návrh spojů na deskách tištěných spojů (PCB - printed circuit board). Tyto dva problémy mají některé společné rysy, zároveň se v některých směrech liší. Oba problémy se dají formulovat jako vyhledávání cesty v pravoúhlém bludišti tvořeném překážkami v podobě součástek a spojů (z angličtiny *routování*). Při návrhu PCB se obvykle pracuje s více vrstvami spojů, zároveň probíhá vyhledávání cest pro velké množství spojů najednou a doba po kterou program problém řeší může být velmi dlouhá (minuty, hodiny, ...). Naproti tomu v našem případě pracujeme jen s jednou vrstvou. Po většinu času se jedná o přidávání jednotlivých nových spojů do stávajícího schématu. Pouze v případech posunu připojených součástek je vyhledáváno více spojů. Časové omezení na vyhledání spoje je mnohem přísnější, protože jeho trvání nesmí uživatele obtěžovat. Měla by být tedy zajištěna odezva, tedy doba vyhledání spoje, v řádu setin nebo desetin sekund. Tyto problémy se také liší v nárocích na tvar výsledného spoje. Zatímco při návrhu tištěných spojů musí cesty splňovat určitá kritéria, která se týkají elektrických vlastností a výrobních možností, spoje v logických schématech by měla splňovat určité nároky na přehlednost, které jsou ovšem do značné míry otázkou subjektivního názoru. Přes tyto rozdíly se ovšem pro řešený problém dají využít algoritmy, používané právě pro návrh PCB.

2.2 Cíl práce

Požadavky na výslednou podobu programu se dají shrnout do těchto bodů:

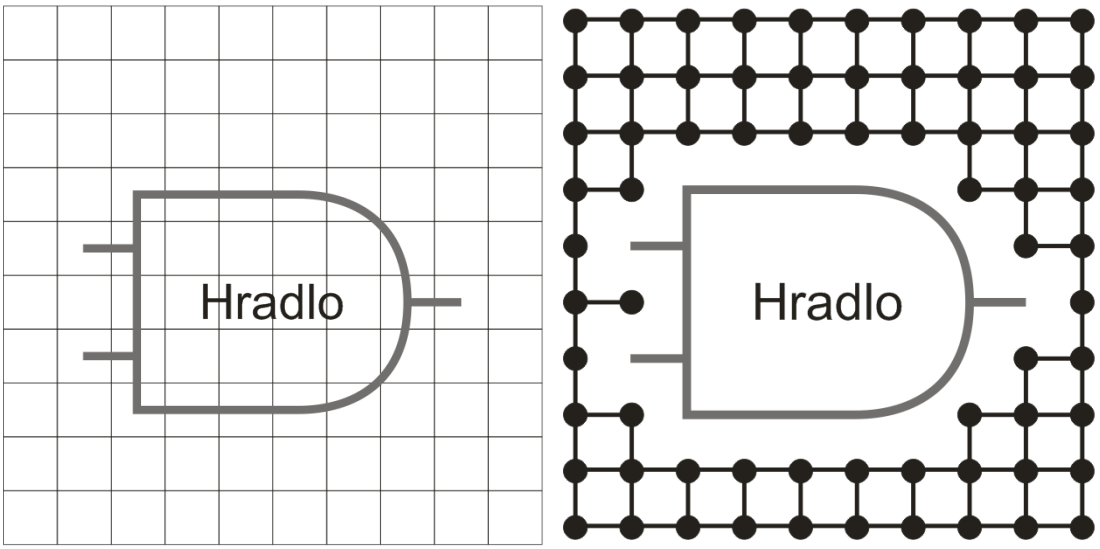
- Umožnit uživateli vytvořit nový spoj pouhým označením počátečního a koncového bodu
- Vytvářet spoje tak, aby by byly co nejpřehlednější
- Zajistit korektní znovu-propojení vodičů při posunu hradel
- Omezit možnosti spojování objektů tak, aby byly dodrženy zásady kreslení schémat
- Zajistit rychlou odezvu

Součástí práce je zároveň vytvořit určitý rešeršní přehled algoritmů, které byly při řešení problému zkoumány.

3 Přehled algoritmů

3.1 Úvod

Tato kapitola si klade za cíl uvést charakteristiku několika základních algoritmů uvažovaných pro řešení našeho problému. Problém vyhledávání cesty v pravoúhlém bludišti lze obecně přeformulovat na problém vyhledávání posloupnosti hran v grafu. Pravoúhlé bludiště lze převést na graf jednoduchým způsobem. Každá buňka bludiště odpovídá jednomu uzlu grafu. Mezi uzlem u odpovídající buňce U a uzlem v odpovídajícím buňce V existuje hrana právě tehdy, pokud buňka V leží tak, že se buňky U dotýká zleva, shora, zprava nebo zdola (obrázek 3.1). Otázka je, jak modifikovat graf tak, aby v sobě zahrnul existující překážky v bludišti. Jednou z možností je nevytvářet (ignorovat) hranu z uzlu u do uzlu v, pokud buňka odpovídající uzlu v je obsazena překážkou (vodič, hradlo...). Druhá možnost je přiřazení čísla hraně (ohodnocení) mezi uzly u a v podle toho, jestli buňka odpovídající uzlu v volná nebo obsazena a jak. Výsledná cena cesty pak může být dána součtem cen jednotlivých hran. Obecně řečeno cenová funkce $f(p)$, je funkce, která cestě(posloupnosti hran) p, přiřazuje reálné číslo(cenu). Cestu mezi uzly u a v, pro kterou platí, že neexistuje žádná jiná cesta mezi u a v s nižší hodnotou $f(p)$, říkáme nejlevnější cesta. Pokud platí, že ohodnocení všech hran je stejné, pak lze tento lze tento pojem sjednotit s pojmem nejkratší cesta, tedy cesta která prochází nejmenším počtem uzlů. Cenová funkce musí splňovat kritérium *konzistence cesty*. To říká, že spojení libovolné nejlevnější cesty p mezi uzly u a v a libovolné nejlevnější cesty q mezi v a w musí být nejlevnější cesta mezi u a w[2].
Uvedeny jsou i algoritmy, které nepracují s grafy, ale jsou založeny na odlišných principech.



Obrázek 3.1: Přechod od buněk bludiště k uzlům grafu

3.2 Dijkstrův algoritmus

Dijkstrův algoritmus, který byl navržen v roce 1959 Edsgerem Dijkstrou, slouží pro nalezení nejlevnější cesty v grafu s nezáporným ohodnocením hran[6].

Postup algoritmu:

1. Vytvoř pole vzdáleností D, pole předchůdců P a množinu uzlů Q.
2. Nastav všechny vzdálenosti v D na nekonečno, kromě startovního uzlu, který nastav na 0.
3. Vlož všechny uzly grafu do Q.
4. Nastav všechny uzly v P na hodnotu $null$(žádný uzel).
5. Vyjmi z Q uzel u s nejmenší hodnotou $D[u]$.
6. Pokud u je koncový uzel jdi na krok 8.
7. Pro všechny sousedy v uzlu u: pokud $D[u]$ + cena mezi u a v <$D[v]$, tak nastav $D[v]$:=$D[u]$ + cena mezi u a v, $P[v] := u$. Jdi na 5.
8. Vytvoř cestu C.
9. Přidej u na konec C.
10. Nastav $u := P[u]$, pokud $u = null$, ukonči algoritmus, jinak jdi na 9.

Časová složitost závisí na implementaci množiny Q. Pokud je implementována jako pole, tak operace vybírání v bodě 5 má lineární složitost a složitost celého algoritmu činí $O(|V|^2 + |E|)$, kde $|V|$ je počet uzlů a $|E|$ počet hran. To znamená, že pro pravoúhlé bludiště, kde $|E| = 4|V| = M \times N$ (M, N jsou rozměry bludiště), je složitost $O(M^2N^2)$. V případě efektivnější implementace Q pomocí binární haldy je složitost $O(MN \log(MN))$.

3.3 A* algoritmus

Algoritmus A* vznikl v roce 1968. Jeho autoři jsou Peter Hart, Nils Nilsson a Bertram Raphael. Jedná se o takzvaný informovaný algoritmus, protože při prohledávání cest jako první prohledává ty, které se zdají být blíže cílovému uzlu. Priorita je dána funkcí $f(p) = g(p) + h(p)$. Funkce $g(p)$ udává cenu cesty. Funkce $h(p)$ je takzvaná odhadová funkce, která odhaduje jak daleko je poslední uzel cesty od řešení[7].

Postup algoritmu:

1. Vytvoř množinu uzlů $CLOSED$ a množinu cest Q.
2. Vlož do Q cestu obsahující jen startovní uzel.
3. Pokud je Q prázdná, ukonči algoritmus. Řešení nebylo nalezeno.
4. Vyber z Q cestu p s nejmenší hodnotou $f(p)$.

5. Přidej poslední uzel u cesty p do množiny $CLOSED$.

6. Pokud uzel u je cílový uzel, ukonči algoritmus. Řešení se nachází v p.

7. Pro všechny sousední uzly v uzlu u: pokud v není v $CLOSED$, přidej cestu p rozšířenou o uzel v do Q. Jdi na 3.

Pro to, aby cesta nalezená algoritmem A* byla optimální, je nutné aby funkce $h(p)$ byla *přípustná*. To znamená, že odhadnutá cena do cíle není nikdy vyšší, než, cena výsledné cesty. Abychom mohli využít množinu $CLOSED$, musí být zároveň splněn požadavek *monotonie*. Pro následníka p_2 cesty p_1(p_2 je rozšířením p_1) musí platit $g(p_1) + h(p_1) \leq g(p_2) + h(p_2)$.

3.4 Leeův algoritmu

Tento algoritmus vznikl v roce 1961 a jeho autorem byl C. Y. Lee. Jedná se o jeden z nejpopulárnějších algoritmů pro řešení routovacího problému. Využívá se zejména při vytváření cest pro vodiče na deskách tištěných spojů. Leeův algoritmus garantuje nalezení cesty, pokud existuje a dále, že tato cesta bude mít minimální cenu[4].

Postup algoritmu[2]:

1. Vyber startovní uzel s, přiřaď uzlu s hodnotu $d(s) = 0$. Přidej uzel s do seznamu L

2. Pro všechny uzly u ze seznamu L s nejmenší hodnotou $d(u)$ najdi sousední přípustný uzel v. Odeber uzel u ze seznamu L. Pokud uzel v nemá přiřazenu hodnotu $d(v)$, přiřaď mu hodnotu $d(v) := d(u) + c$, kde c je cena hrany mezi u a v a přidej ho do seznamu L_1. Pokud uzel v je cílovým uzlem, přejdi na krok 6.

3. Přidej uzly ze seznamu L_1do seznamu L. Vymaž seznam L_1.

4. Odeber všechny uzly z L, jejichž sousedé v mají přiřazenu hodnotu $d(v)$.

5. Pokud seznam L není prázdný, jdi na krok 2. V opačném případě ukonči algoritmus, řešení *nebylo* nalezeno.

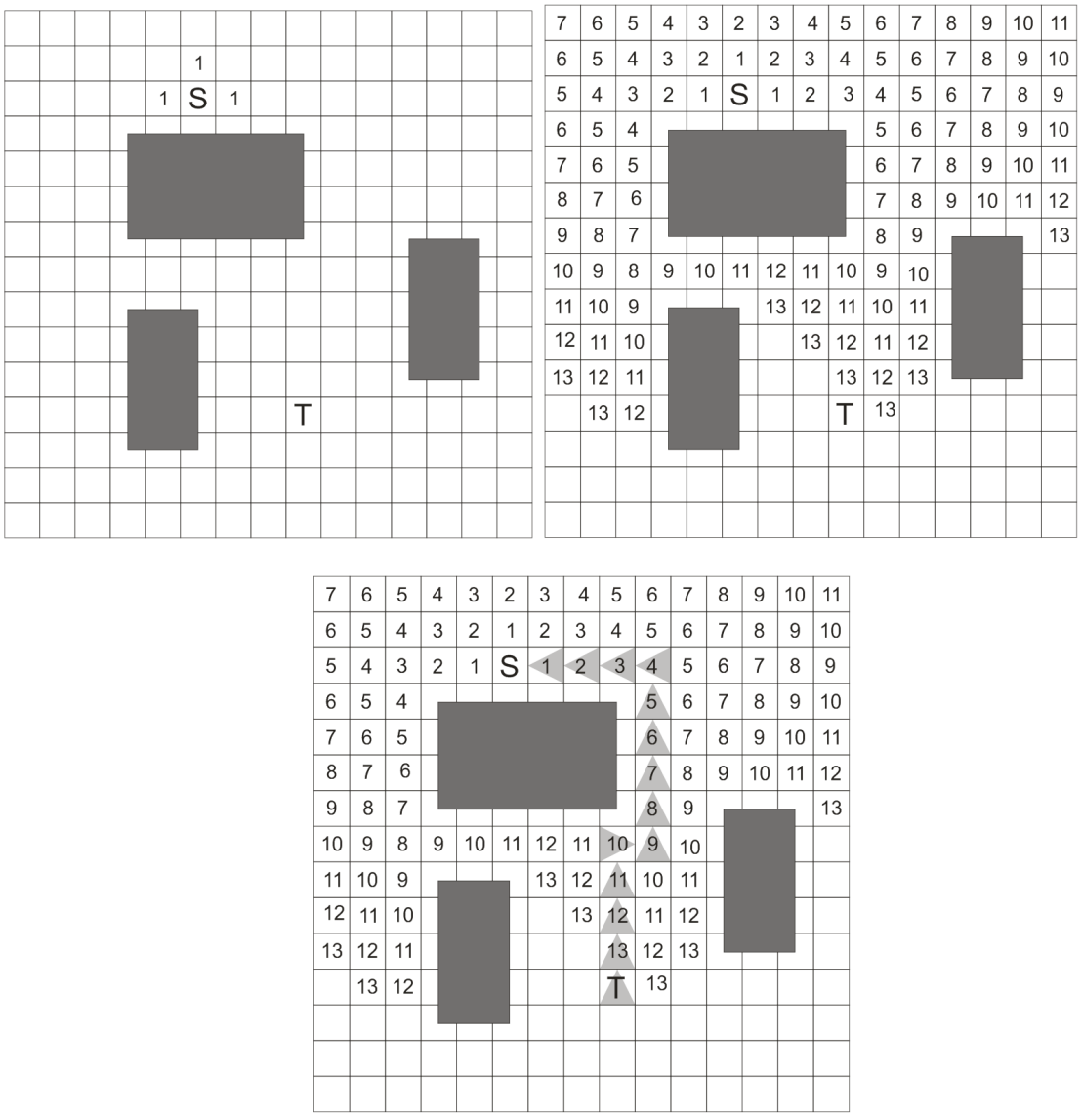
6. Přiřaď $u := t$, kde t je cílový uzel. Inicializuj cestu C.

7. Najdi souseda v uzlu u, který má nejnižší hodnotu $d(v)$.

8. Přidej uzel v do cesty C. Pokud uzel v je startovní uzel, ukonči algoritmus, řešení *bylo* nalezeno. V opačném případe jdi na krok 6.

Modifikace algoritmu Pokud se omezíme na hodnotu hrany $f(h)$ vždy rovno 1, zjistíme že hodnoty sousedních buněk u a v se vždy liší tak, že $d(u) = d(v) \pm 1$. Tohoto jevu lze využít ke snížení paměťové náročnost kódování stavu buňky. Pro označení hodnoty stačí používat čísla 0, 1 a 2. Vzorec v bodě 2 bude tedy vypadat $d(v) := (d(u) + 1) \bmod 3$. Dále je potřeba zakódovat stavy *neinicializovaný* a *blokovaný*. Toto tedy vede na 5 různých stavů, které lze kódovat pomocí 3 bitů.

Vlastnosti V nejjednodušším případě, kdy $f(h)$ je vždy rovno jedné se v podstatě jedná o algoritmus BFS, tedy vyhledávání do šířky. Postup algoritmu v tomto případě vidíme na obrázku 3.2. Nevýhodou algoritmu je nemožnost zahrnutí počtu ohybů do ceny cesty.

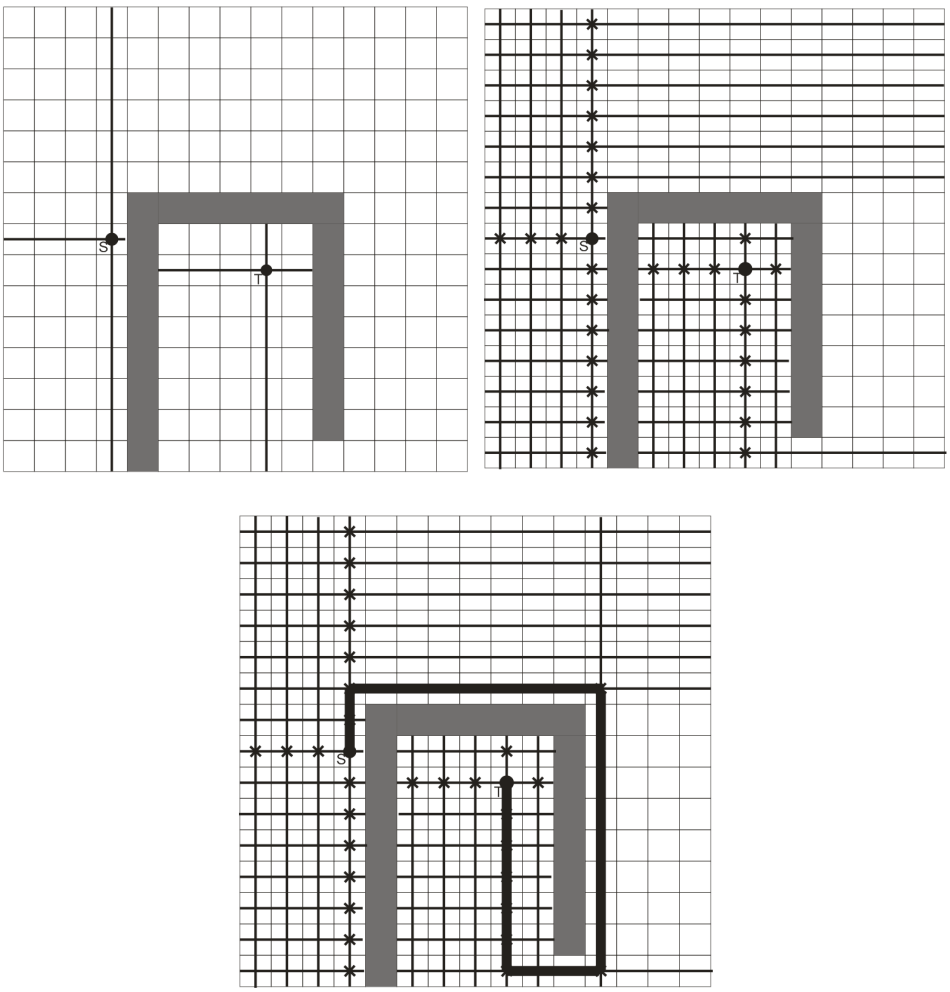


Obrázek 3.2: Postup Leeova algoritmu

3.5 Mikami-Tabuchi

Algoritmus Mikami-Tabuchi na rozdíl od doposud uvedených algoritmů není založen na prohledávání grafu. Jeho principem je generování úseček ze startovního a cílového bodu, které tvoří tvar výsledné cesty. Algoritmus začíná vytyčením horizontálních a vertikálních úseček ze startovního bodu a z cílového bodu. Pokud se úsečky protnou, tak jsou použity jako hledané řešení. Úsečky se nemusí protnout, protože jsou omezeny nejbližší překážkou. Pokud se neprotnou, jsou generovány nové úsečky, kolmo na stávající v takzvaných *escape* bodech. Tyto body se nacházejí ve středech buněk, kterými úsečky procházejí. Tento postup je opakován dokud nedojde k protnutí úseček vedených ze startovního a z cílového bodu[3, 4].

Vlastnosti Složitost tohoto algoritmu je dána $O(L)$, kde L je počet vygenerovaných úseček. Tento algoritmus zaručuje nalezení cesty pokud existuje, ale nezaručuje že tato cesta bude nejkratší.



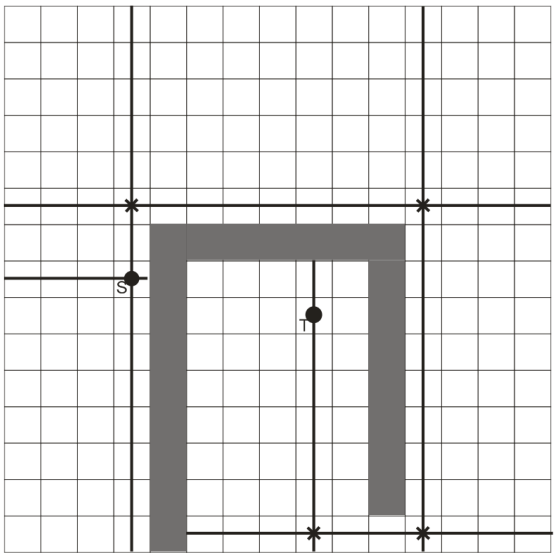
S značí startovní bod, T cílový bod. Křížek označuje escape bod.

Obrázek 3.3: Algoritmus Mikami-Tabuchi

3.6 Hightower

Algoritmus Hightower pracuje na stejném principu jak Mikami-Tabuchi. Liší se ve způsobu generování escape bodů. Zatímco u Mikami-Tabuchi je počet těchto bodů dán počtem buněk, které úsečka protíná, u Hightowerova algoritmu je generován jen jeden escape bod na úsečku. Tento bod je umisťován hned za překážku, která vede rovnoběžně s původní přímkou[3, 4].

Vlastnosti Na rozdíl od algoritmu Mikami-Tabuchi, tento algoritmus nezaručuje vždy nalezení cesty, přestože může existovat.



S značí startovní bod, *T* cílový bod. Křížek označuje escape bod.

Obrázek 3.4: Algoritmus Hightower

4 Analýza a návrh řešení

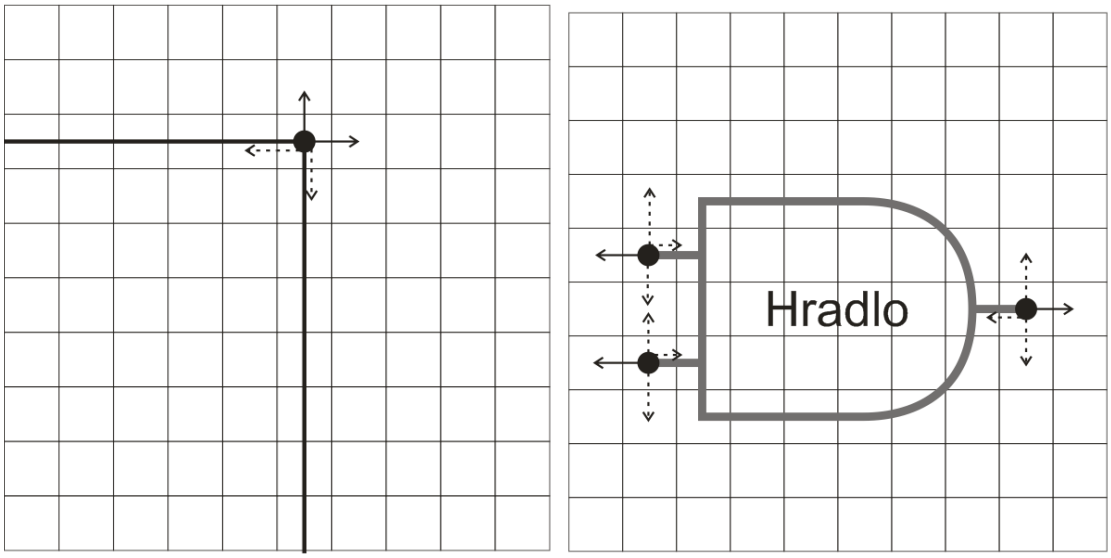
V této kapitole uvedeme nejdříve nutné modifikace stávajícího programu, které s implementací propojování souvisí. Dále je pak uveden samotný routovací algoritmus.

4.1 Spojové uzly

Původní program předpokládal, že routovací funkce bude mít podobu funkce, která bude mít jako parametry dva body v bludišti a nějakým způsobem bude schopna najít vhodnou cestu pro spoj, kterou vrátí v podobě seznamu rohových bodů na této cestě. Původní jednoduchá routovací funkce byla tímto způsobem také implementována. Nicméně tento přístup se během mé práce ukázal jako nevýhodný. Problémem bylo, že tato funkce ve své podstatě nevěděla, které objekty se snaží propojit, nebyla schopna určit preferovaný směr připojení spoje (u hradla směrem od něj), nedokázala rozpoznat stávající spojení dvou vodičů, kterému je potřeba se vyhnout.
Tento problém jsem řešil zavedením nového grafického objektu - spojového uzlu. Tento objekt je reprezentací vodivého spojení mezi vodiči. Jedná se v podstatě o bod, ke kterému může být připojen spoj. Spojový uzel umožňuje připojení až 4 různých spojů, přičemž každý musí být připojen z jiné strany (zleva, shora, zprava, zdola). Spojový uzel se může také nalézat na místě vstupu nebo výstupu hradla (případně bloku). V tomto případě umožňuje přípojení jen jednoho spoje a to ze směru od hradla (směrem ven)(obrázek 4.1). Původní program umožňoval připojovat spoje přímo k definovaným bodům na hradlech, blocích nebo ke stávajícím vodičům. V mém řešení jsem se rozhodl připojovat spoje jen k těmto spojovým uzlům. Toto má za následek sjednocení procesu routování spoje pro všechny případy propojování (hradlo-hradlo, hradlo-spoj, spoj-spoj). V případě propojování dvou vstupů/výstupů hradel tedy jde o propojování dvou již existujících spojových uzlů. Pokud jde o připojování z existujícího spojového uzlu k vodiči, je nejprve tento vodič v místě připojování rozdělen na dva a v tomto místě je vytvořen nový spojový uzel. Routování pak i v tomto případě probíhá mezi těmito dvěma uzly. Další výhodou využití spojových uzlů je možnost omezení počtu spojů v určitém bodě. Program neumožňuje připojení k uzlu, který má vyčerpány všechny své přípojné směry.

4.2 Uložení objektů

Předpokladem implementace routovacího algoritmu je přítomnost rozhraní v programu sloužícího k efektivnímu vyhledávání objektů (tedy případných překážek) na daných souřadnicích.
Pro potřeby uložení prostorového rozložení objektů slouží v původním projektu systém registrace objektů na základě jejich prostorových souřadnic a rozměrů. V podstatě se jedná o dvě pole spojových seznamů. Jedno pro horizontální osu a druhé pro osu vertikální. Velikost pole je dána počtem buněk v bludišti v daném směru. Určitý spojový seznam obsahoval záznam o objektu právě tehdy, pokud objekt svými rozměry zasahoval v daném směru na odpovídající souřadnici. Toto umožňuje velmi efektivní implementaci metody hit-test, tedy zjištění objektu na zadaných souřadnicích. Určitou výjimku ovšem tvoří spoje, které v původním projektu byly registrovány jen v místech svých rohových bodů. Toto způsobuje určité zpomalení při zjišťování vodičů v daném bodě a zejména



Puntíky znázorňují spojové uzly. Plné šipky znázorňují volné směry připojení. Čárkované šipky znázorňují obsazené směry.

Obrázek 4.1: Spojové uzly

zjišťování jejich směru. Protože tento systém v původním projektu sloužil prakticky jen pro implementaci vybírání objektů při kliknutí uživatele, tak tato komplikace z pohledu časové náročnosti nehrála roli. Pro efektivní vyhledávání objektů pro potřeby routovacího algoritmu bylo však potřeba tento systém pro všechny objekty sjednotit. Systém registrace objektů byl tedy upraven tak, aby spoj byl registrován ve všech svých bodech. Zároveň je také ve spojových seznamech zahrnuta informace o směru vodiče v daném bodě.

4.3 Routovací algoritmus

Základem mého řešení je algoritmus A*. Pro tuto variantu jsem se rozhodl intuitivně na základě předchozích zkušeností s implementací tohoto algoritmu. Při návrhu konkrétní varianty algoritmu bylo zejména potřeba vyřešit otázku vhodné cenové funkce $g(p)$ a odhadové funkce $h(p)$. Jako nejvhodnější odhadová funkce je na první pohled takzvaná Manhattanská vzdálenost[1] mezi koncový uzlem cesty p a cílovým uzlem. Manhattanská míra definuje vzdálenost mezi dvěma body P a Q jako $d(P,Q) = |P_x - Q_x| + |P_y - Q_y|$. Tato funkce splňuje podmínku přípustnosti, protože odhadovaná vzdálenost je vždy stejná nebo menší jako cena nalezené cesty (což vyplývá z vlastnosti použité cenové funkce, jak uvidíme později). Dále požadujeme vlastnost monotonie heuristické funkce $f(p) = g(p) + h(p)$. Dokažme si nejdříve, že tuto vlastnost splňuje odhadová funkce $h(p)$. Monotonie (trojúhelníková nerovnost) říká, že pro libovolnou trojici tří bodů P, Q, R platí $d(P,Q) + d(Q,R) \geq d(P,R)$. Ukažme si, že tuto vlastnost Manhattanská míra splňuje.

[1]Pokud se dále v textu vyskytuje jen pojem *vzdálenost* je tím implicitně myšlena Manhattanská vzdálenost.

4.3.1 Důkaz monotonie Manhattanské míry

Označme si souřadnice bodů P jako x_0, y_0, Q jako x_1, y_1 a R jako x_2, y_2. Nerovnici

$$d(P,Q) + d(Q,R) \geq d(P,R)$$

poté můžeme zapsat jako

$$|x_0 - x_1| + |y_0 - y_1| + |x_1 - x_2| + |y_1 - y_2| \geq |x_0 - x_2| + |y_0 - y_2|$$

Využijeme toho, že můžeme trojici bodů posunout o libovolný vektor a otočit a násobek pravého úhlu beze změny vzájemných vzdáleností. Tato vlastnost vychází ze zřetelné osové symetrie. Předpokládejme tedy bez újmy na obecnosti, že

$$x_0 = 0, y_0 = 0$$

$$x_2 \geq 0, y_2 \geq 0$$

x_1a $y_1$mohou nabývat libovolných hodnot z intervalu $(-\infty, \infty)$. Nerovnost potom bude mít tvar

$$|x_1| + |y_1| + |x_1 - x_2| + |y_1 - y_2| \geq x_2 + y_2$$

Tuto nerovnost je nutné dokázat pro všechna x_1a y_1. Důkaz rozdělíme pro hodnoty x_1 v intervalech $x_1 \in (-\infty, 0)$, $x_1 \in \langle 0, x_2)$a $x_1 \in \langle x_2, \infty)$ a obdobně pro hodnoty y_1. Toto tedy vede na 9 samostatných důkazů.

1. $x_1 < 0, y_1 < 0$

$$-x_1 + -y_1 + (x_2 - x_1) + (y_2 - y_1) \geq x_2 + y_2$$
$$-2x_1 - 2y_1 + x_2 + y_2 \geq x_2 + y_2$$
$$-2x_1 - 2y_1 \geq 0$$

2. $0 \leq x_1 < x_2, y_1 < 0$

$$x_1 + -y_1 + (x_2 - x_1) + (y_2 - y_1) \geq x_2 + y_2$$
$$-2y_1 + x_2 + y_2 \geq x_2 + y_2$$
$$-2y_1 \geq 0$$

3. $x_1 < 0, 0 \leq y_1 < y_2$ analogicky k 2.

4. $0 \leq x_1 < x_2, 0 \leq y_1 < y_2$

$$x_1 + y_1 + (x_2 - x_1) + (y_2 - y_1) \geq x_2 + y_2$$
$$x_2 + y_2 \geq x_2 + y_2$$
$$0 \geq 0$$

5. $x_1 \geq x_2, y_1 < 0$

$$x_1 - y_1 + (x_1 - x_2) + (y_2 - y_1) \geq x_2 + y_2$$

$$2x_1 - x_2 - 2y_1 + y_2 \geq x_2 + y_2$$

Protože $x_1 > x_2$, dá se $x_1$vyjádřit jako $x_1 = x_2 + d_x$, kde $d_x \geq 0$. Potom

$$2x_2 + 2d_x - x_2 - 2y_1 + y_2 \geq x_2 + y_2$$

$$2d_x - 2y_1 \geq 0$$

6. $x_1 < 0, y_1 \geq y_2$ analogicky k 5.

7. $x_1 \geq x_2, 0 \leq y_1 < y_2$

$$x_1 + y_1 + (x_1 - x_2) + (y_2 - y_1) \geq x_2 + y_2$$

$$2x_1 - x_2 + y_2 \geq x_2 + y_2$$

$$2x_2 + 2d_x - x_2 + y_2 \geq x_2 + y_2$$

$$2d_x \geq 0$$

8. $0 \leq x_1 < x_2, y_1 \geq y_2$ analogicky k 7.

9. $x_1 \geq x_2, y_1 \geq y_2$

$$x_1 + y_1 + (x_1 - x_2) + (y_1 - y_2) \geq x_2 + y_2$$

$$2x_1 + 2y_1 - x_2 - y_2 \geq x_2 + y_2$$

$y_1$vyjádříme jako $y_1 = y_2 + d_y$, $d_y \geq 0$

$$2x_2 + 2d_x + 2y_2 + 2d_y - x_2 - y_2 \geq x_2 + y_2$$

$$2d_x + 2d_y \geq 0$$

Trojúhelníková nerovnost tedy platí.

4.3.2 Cenová funkce

Cenová funkce slouží pro určení ceny cesty. Algoritmus A* zaručuje nalezení cesty s nejmenší hodnotou této ceny, proto veškeré nároky na kvalitu nalezené cesty je nutné formalizovat pomocí této funkce. Byla stanovena tato kritéria na vhodnou cestu (od nejmenší důležitosti po největší):

- Co nejmenší počet buněk (co nejkratší cesta)
- Co nejmenší počet ohybů cesty
- Co nejmenší počet křížení s existujícími spoji
- Žádná křížení s plošnými překážkami (hradla, ...)

Křížení spojů je nutné rozdělit na tři samostatné případy. Jedná se o samotné křížení, kdy nový spoj protíná v buňce stávající spoj pod pravým úhlem, dále případ, kdy oba spoje v buňce míří rovnoběžně a konečně když jeden nebo oba ze spojů v buňce mění svůj směr. Zatímco první případ nepředstavuje žádný problém, dalším dvěma případům křížení je dobré se vyhnout. Výsledná funkce ohodnocující cestu p by tedy měla mít tvar $g(p) = d \cdot D + t \cdot T + c_c \cdot C_c + c_p \cdot C_p + c_t \cdot C_t + s \cdot S$. Význam jednotlivých parametrů uvádí tabulka 4.1.

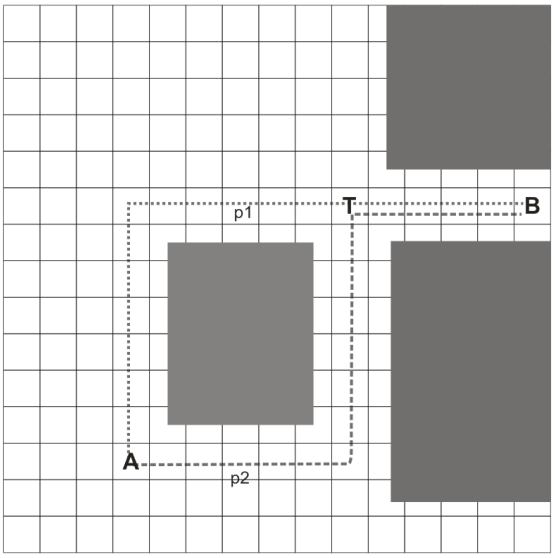
Parametr	Význam	Konstanta ceny
d	Počet buněk v cestě	D
t	Počet ohybů cesty	T
c_c	Počet kolmých křížení s existujícími spoji	C_c
c_p	Počet rovnoběžných setkání s existujícími spoji	C_p
c_t	Počet křížení s ohybem spoje	C_t
s	Počet buněk v cestě zasahujících do plošných překážek	S

Tabulka 4.1: Parametry cenové funkce

Konstanty D,T,C_c,C_p,C_t,S určují cenu jednotlivých případů. Vzhledem k určeným prioritám jednotlivých kritérií by mělo platit $D < T < C_c < C_p < C_t < S$. Pro zaručení přípustnosti heuristické funkce nesmí být nikdy cena nejkratší možné cesty mezi dvěma body větší než odhad. To znamená, že je potřeba zvolit D tak, aby pro nejkratší možnou cestu p_{min} mezi body (buňkami) A a B, pro niž platí, že kromě d jsou ostatní parametry nulové, platilo $d \cdot D \geq |A_x - B_x| + |A_y - B_y|$, D tedy musí být větší nebo rovno rozměru buňky, který činí 10. Hodnoty ostatních konstant je možné nastavit na základě experimentů.

Takováto funkce má ovšem závažný nedostatek v tom, že porušuje požadavek na konzistenci cesty. Představme si případ, kde chceme vést cestu mezi body A a B. Před bodem B se nachází jakýsi tunel z překážek. Vstup do tohoto tunelu označme T. Předpokládejme že mezi body A a T lze najít dvě cesty p_1 a p_2 se stejným ohodnocením. Mezi body A a T se nenachází žádná cesta s menším ohodnocením. Cestu mezi T a B označme q. Cesta p_1 v bodě T vede ve se směru průchodu tunelem, cesta p_2 kolmo ke vstupu. Pravidlo o konzistenci cest říká, že spojení minimálních cest mezi body A a T a body T a B musí být minimální cestou mezi body A a B. Toto ovšem splňuje jen cesta p_1. Při propojování z T do B po cestě p_2 je nutné udělat o jeden ohyb víc. V průběhu výpočtu algoritmu se to projeví tak, že když se uzel odpovídající bodu T vyjme z prioritní fronty a nastaví na uzavřený, může mít nastaven jako svého předchůdce uzel ležící na cestě p_2. Přestože by nakonec měla být vybrána cesta p_1, tak uzel T už znovu nebude otevřen a algoritmus zůstane u cesty p_2(obrázek 4.2).

Tento problém jsem se rozhodl vyřešit modifikací zobrazení množiny buněk na množinu uzlů. V této modifikaci každé buňce odpovídají dva různé uzly grafu. Tyto uzly se liší směrem svého předchůdce. Jinými slovy to znamená rozdělení množiny *CLOSED* na uzly uzavřené v horizontálním směru a uzly uzavřené ve vertikálním směru (obrázek 4.3). Vzhledem k tomu, že cena cesty zbývajícího úseku závisí vždy jen na posledním směru ohybu cesty, tak tato modifikace postačuje pro splnění podmínky konzistence cesty. Nevýhodou plynoucí z této modifikace je zdvojnásobení prohledávaného prostoru.

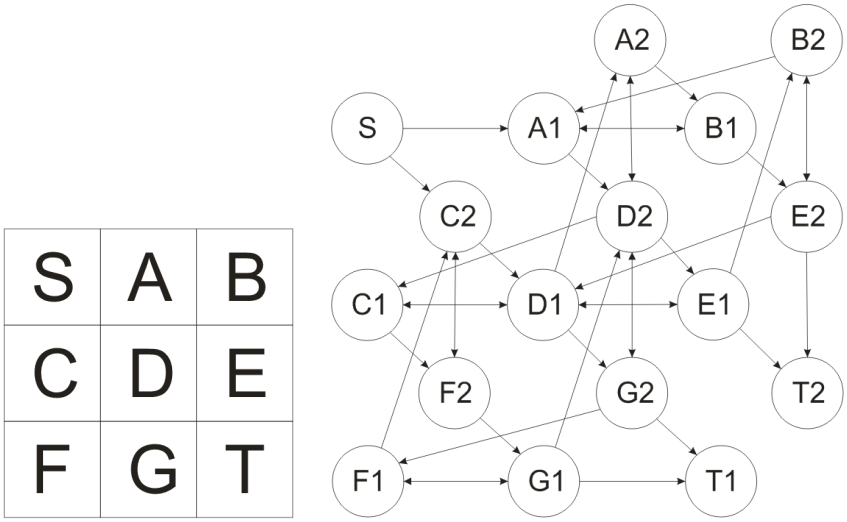


Obrázek 4.2: Příklad chyby konzistence cesty

4.4 Škálovatelnost

Časová náročnost algoritmu závisí na rozměrech bludiště, vzdálenosti startovního a cílového bodu hledané cesty a množství překážek v bludišti. V případě hledání cesty v prázdném bludišti se naplno projeví efekt odhadové funkce. To znamená, že počet uzlů vybraných z prioritní fronty bude roven Manhattanské vzdálenosti startovního a cílového bodu vydělené rozměrem buňky a tedy zároveň roven počtu uzlů výsledné cesty. Spodní hranice časové náročnosti se dá tedy vyjádřit jako $O(d \log d)$, kde d označuje vzdálenost startovního a cílového bodu, v maximálním případě tedy $d = M + N$ (M,N označují rozměry bludiště v ose x a y). V případě vyššího počtu překážek v bludišti, které však neovlivní výslednou cestu, se projeví časová náročnost metody hit-test, která slouží pro "osahání" pozice v bludišti při výpočtu ceny. Náročnost této metody je asymptoticky dána $O(\log n)$, kde n je počet překážek v bludišti. Náročnost celého algoritmu tedy v tomto případě nabývá tvar $O(d \log d \cdot \log n)$. V případě velmi zaplněného bludiště v oblasti mezi startovním a cílovým bodem přestává odhadová funkce efektivně ovlivňovat průběh algoritmu a počet uzlů vyjmutých z fronty roste rychleji než lineárně se vzdáleností bodů. V nejhorším případě to můžou být(teoreticky) až všechny body v bludišti. V takovém případě je pak složitost dána $O(MN \log(MN) \log n)$. Pokud budeme předpokládat, že bludiště má vždy víceméně čtvercový tvar, můžeme počítat jen s rozměrem v jedné ose, který označíme S. V obecném případě pak můžeme náročnost vyjádřit jako $O(S^f \log S^f \cdot \log n)$, kde exponent f nabývá hodnot z intervalu $\langle 1, 2 \rangle$ a v principu vyjadřuje odlišnost výsledné cesty od cesty přímé.
Jako jeden z požadavků na vlastnosti automatického propojování byla uvedena rychlá odezva. To znamená, že výsledek by měl být dosažen v rámci desetin sekundy. Program využívá bludiště o rozměrech 100x200 buněk. Při testování[2] propojování na vzdálenost protilehlých rohů bludiště s rozumným zaplněním dosahovala doba výpočtu řádově těchto hodnot. Dá se tedy říci, že zvolené řešení je použitelné maximálně pro bludiště podobných rozměrů.

[2]Testovací počítač používá CPU Intel Core Duo 1,6GHz



1 na konci označuje uzel s horizontálním předchůdcem, 2 s vertikálním předchůdcem.

Obrázek 4.3: Přechod od buněk bludiště k uzlům grafu.

4.5 Techniky pro zvýšení výkonu

Pro určité zvýšení výkonu algoritmu je využito několika technik, používaných v oblasti návrhu spojů na PCB.

Uspořádání bodů Při hledání cesty mezi dvěma spojovými uzly, je jako startovní nastaven ten, který má nižší počet neobsazených směrů. Pokud mají oba spojové uzly stejný počet volných směrů, je jako startovní nastaven ten, který se nachází dál od středu bludiště.

Uspořádání cest Pokud dojde k požadavku na routování více cest (posun hradla,...), jsou tyto požadavky uspořádány podle vzdálenosti startovního a cílového bodu v pořadí od nejmenší hodnoty.

Omezení prostoru Prohledávané bludiště je při vyhledávání cesty omezeno pomyslným obdélníkem o 20% širším a vyšším, než je obdélník vymezený startovním a cílovým bodem. Pokud v takto omezeném bludišti není cesta nalezena, je postup algoritmu opakován bez tohoto omezení.

5 Realizace

5.1 Úvod

Tato kapitola popisuje vlastní implementaci funkcí do programu. Předpokladem pro korektní implementaci routování spojů byly některé další úpravy, které s tímto problémem nepřímo souvisí. Ve stávajícím programu INKS bylo zejména potřeba upravit vnitřní reprezentaci jednotlivých grafických objektů. S těmito změnami pak úzce souvisí i úpravy v částech programu sloužící pro ukládání a načítání souborů.

5.2 Grafické objekty

V původním programu byly jednotlivé objekty reprezentovány jako instance 5 různých tříd. Jednalo se o třídy *CGate*, *CInv*, *CTag, CBlock* a *CWire*. Třída *CGate* slouží pro reprezentaci hradla (AND, OR, NAND, ...), *CInv* reprezentuje invertor, *CTag* značku označující vstup nebo výstup, *CBlock* součástku s vnitřním schématem a *CWire* slouží pro reprezentaci spoje. Všechny tyto třídy jsou odvozeny od společné třídy *CCommonObject*.
Pro zjednodušení řešení problému byla tato struktura tříd upravena. Byla odstraněna třída *CInv*, protože invertor je ve své podstatě hradlo a lze ho tedy reprezentovat pomocí třídy *CGate*. Třídy *CBlock* a *CTag* jsou nyní odvozeny od třídy *CGate*, což umožňuje na ně pro potřeby routování nahlížet jednotně jako na plošné objekty. Dále byla přidána třída *CNode* pro reprezentaci spojového uzlu.
Pro zjednodušení problému byla odstraněna možnost otočení hradel o libovolný úhel. Úhly otočení byly omezeny na 0°, 90°, 180° a 270°. Toto zaručuje že připojovací uzly hradel se budou vždy nacházet ve středech buněk propojovacího bludiště. Tato možnost byla také zbytečná, protože takto "nakloněná" hradla by neodpovídala zásadám tvorby schémat.
Byla taktéž změněna vnitřní reprezentace fyzického propojení mezi objekty. Původní verze programu umožňovala reprezentovat propojení mezi libovolnými prvky pomocí struktury *Connections*, která byla součástí každého objektu a udržovala pole připojených objektů. Propojení mezi objekty je nyní rozděleno na dva koncepty: *připojení* spojového uzlu k hradlu a *propojení* spojových uzlů pomocí spojů. Připojení uzlu k hradlu je z pohledu spojového uzlu implementováno pomocí ukazatele *CCommonObject *m_pAttachedObject*, který ukazuje na připojený objekt. Z pohledu hradla je udržován přehled připojených spojových uzlů v poli *vector<CNode *> m_PinNodes*. Propojení mezi spojovými uzly a spoji je z pohledu spoje reprezentováno dvojicí ukazatelů *CCommonObject *m_pStartNode, m_pEndNode*, ukazujících na připojené uzly a z pohledu uzlu čtveřicí ukazatelů *CCommonObject *m_pLeft, *m_pTop, *m_pRight, *m_pBottom*, ukazujících na jednotlivé spoje pro každý možný směr připojení.

5.3 Routování

Pro zajištění nízké časové náročnosti algoritmu A* bylo nutné pečlivě zvážit způsob implementace jednotlivých jeho částí. Zejména se jedná o způsob uložení otevřených cest do prioritní fronty a samotná implementace prioritní fronty. Dále implementace množiny *CLOSED* a způsob výpočtu ceny cesty.

Jako prioritní fronta je využita šablonová třída *std::priority_ queue*. Jejím základem je binární halda, pro kterou platí, že operace vyjmutí minimálního prvku a přidání prvku mají složitost $O(\log n)$[5]. Prvky v této frontě jsou ukazatelé na třídu *CRoutingNode*, která zaznamenává souřadnice poslední buňky v cestě, cenu cesty od počátku, odhad ceny do cíle a ukazatel na předchůdce. Množina *CLOSED* je implementována jako pole o velikosti $M \times N$ typu *unsigned char*. Buňky uzavřené v horizontálním směru mají nastaven v tomto poli na odpovídajících souřadnicích nultý bit a buňky uzavřené ve vertikálním směru první bit. Ostatní bity se nepoužívají.
Pro výpočet ceny cesty je nutné zjistit, jestli přidávaná buňka je obsazena a co na ní případně leží. K tomu je využívána třída *CSpacialHashing*, která udržuje informaci o prostorovém rozložení objektů. Jelikož metoda pro zjištění této informace nemá konstantní časovou náročnost, tak se tato informace udržuje v pomocném poli *CostBuffer* a může být znovu využita při opětovném otevření dané buňky. Zároveň se toto pole využívá při routování dalších spojů, pokud se propojuje více spojů. Informace o buňce v tomto poli má podobu čísla od 0 do 7. Význam jednotlivých hodnot uvádí tabulka 5.1.

Hodnota	Význam
0	Neinicializovaná hodnota. Informaci o buňce je nutné zjistit.
1	Buňka je volná.
2	V buňce se nachází plošná překážka - hradlo, blok, ...
3	Buňkou vede vertikální spoj.
4	Buňkou vede horizontální spoj.
5	V buňce se nachází ohyb spoje.
6	Buňka je startovní.
7	Buňka je cílová.

Tabulka 5.1: Hodnoty v poli *CostBuffer*

Tato informace spolu s informací o směru předchozí buňky umožňuje jednoduše vypočítat přírůstek cenové funkce, který je přičten k hodnotě předchůdce. Konstanty cenové funkce byly zvoleny tak, jak uvádí tabulka 5.2. Z těchto hodnot vyplývá snaha omezit počet ohybů a křížení. Cena za kolmé křížení je nastavena jako trojnásobek ceny ohybu, což odpovídá snaze vyhnout se křížení vodičů, ovšem ne za cenu příliš velkého počtu ohybů. Hodnoty ostatních konstant napovídají snahu vyhnout se těmto případům za každou cenu a díky tomu za běžných okolností k těmto případům nedochází.

Konstanta	Význam	Hodnota
D	Cena jedné buňky	10
T	Cena ohybu	20
C_c	Cena kolmého křížení s existujícím spojem	60
C_p	Cena rovnoběžného setkání s existujícím spojem	1000
C_t	Cena křížení na místě existujícího ohybu jiného spoje	1000
S	Cena za křížení s plošnou překážkou	10000000

Tabulka 5.2: Zvolené hodnoty konstant

6 Testování

6.1 Testování funkčnosti

Při testování korektní funkčnosti programu po provedení změn, bylo nutné ověřit všechny aspekty programu. Zejména bylo nutné zaměřit se na jeho modifikované části.

Testované funkce

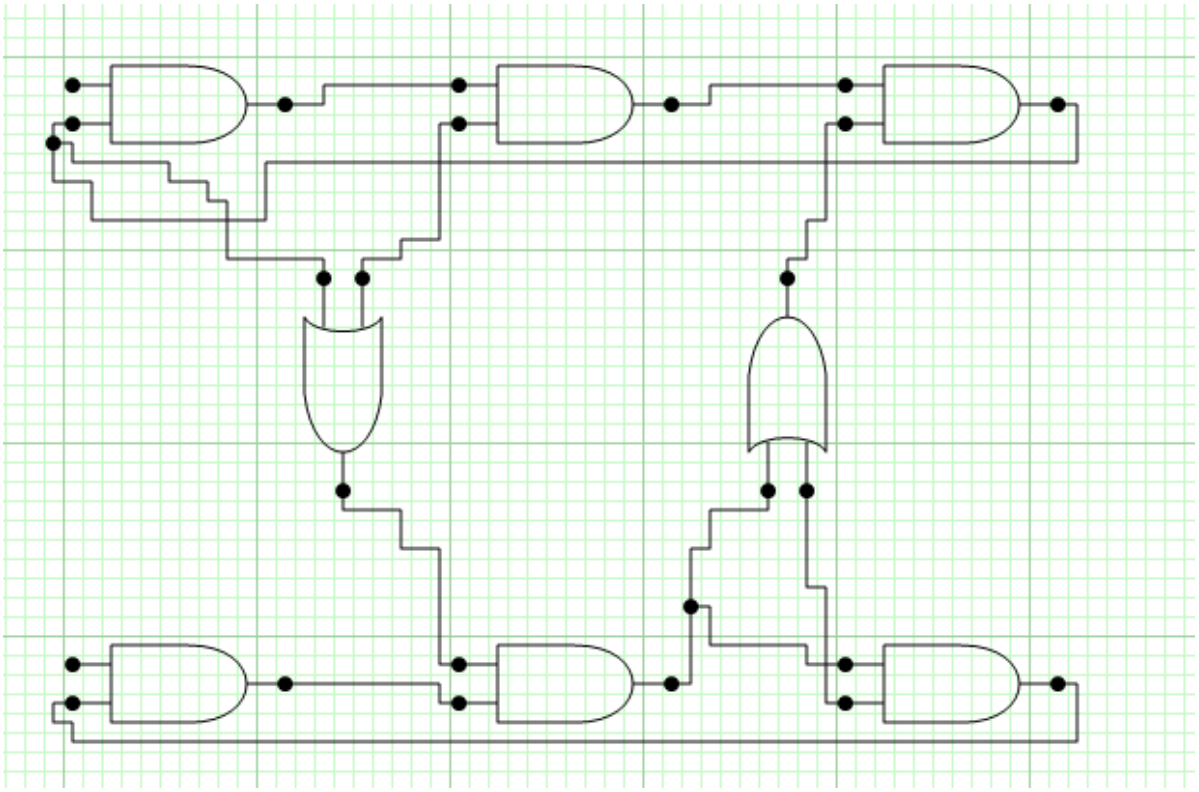
- Propojování hradel
- Posun hradel
- Posun více vybraných objektů
- Ukládání/načítání souborů
- Vytváření bloků

6.2 Modelový případ

Tato část předkládá výsledky získané při propojování spojů na modelovém příkladu s užitím dvou variant hodnot konstant cenové funkce. Zároveň ukazuje výsledek propojování v podobném případě v komerčním programu Multisim 10 od společnosti National Instruments(r).

6.2.1 Cenová funkce bez ohodnocení ohybů a křížení

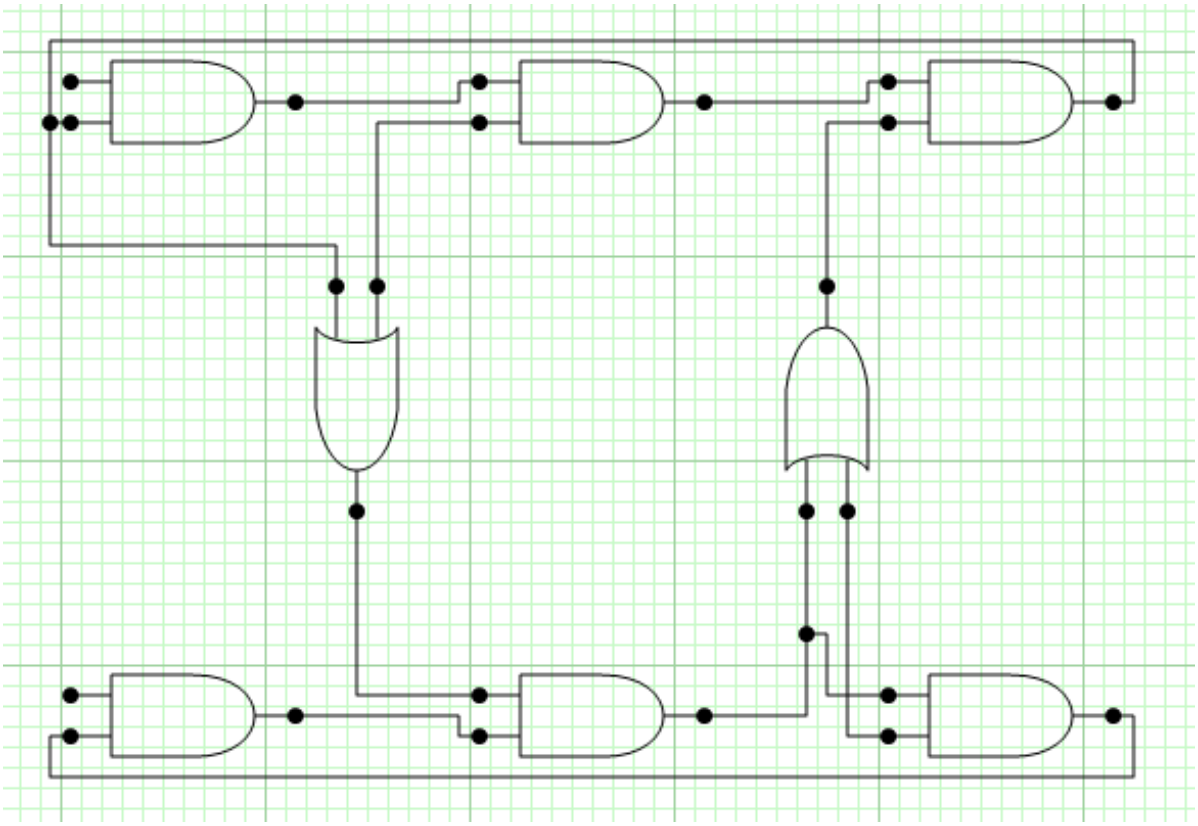
V tomto případě byly zvoleny konstanty tak, že nezohledňovaly počet ohybů a křížení. Konstanty T a C_c byly tedy nastaveny na 0. Je patrné, že vypočtené cesty v tomto případě mají schodovitý tvar. Také cesta vedená mezi prvním a posledním hradlem v horní řadě zbytečně kříží stávající ostatní cesty (ty byly vedeny dříve než tato).



Obrázek 6.1: Výsledek bez ohodnocení ohybů a křížení

6.2.2 Cenová funkce s ohodnocením ohybů a křížení

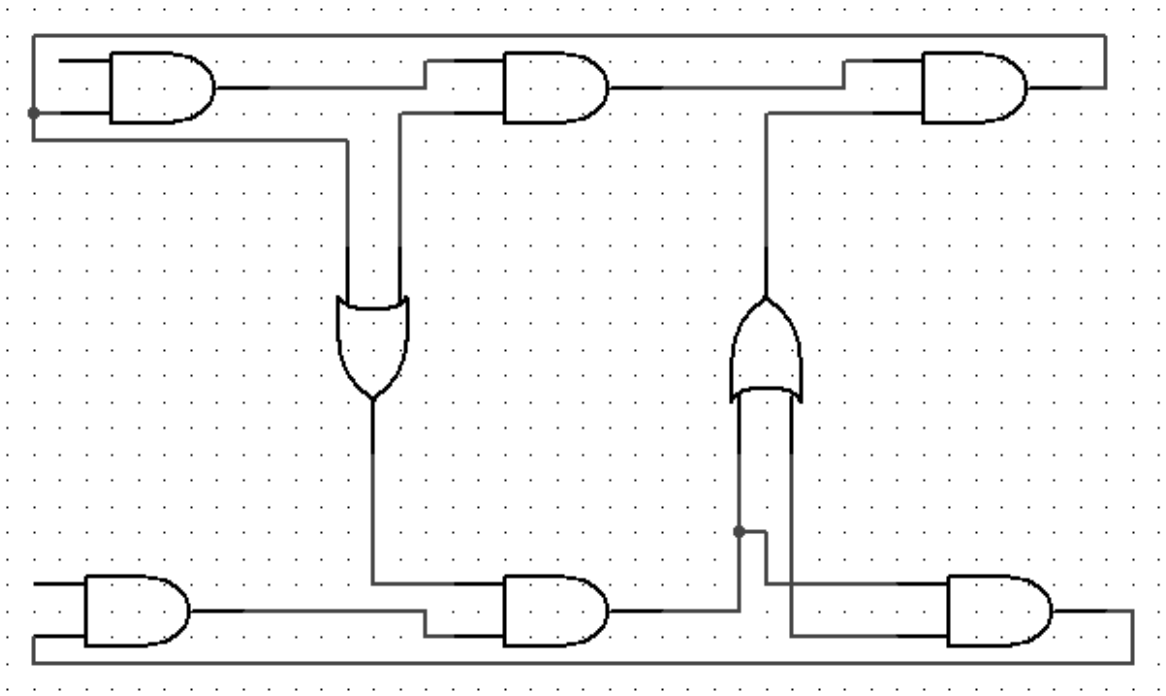
Zde byly konstanty nastaveny tak, jak je uvedeno v tabulce 5.2. Vidíme, že cesty jsou přehledné a mají minimální počet ohybů.



Obrázek 6.2: Výsledek s ohodnocením ohybů a křížení

6.2.3 Výsledek komerčního programu

Výsledek zapojování obdobného schématu v programu Multisim 10 vykazuje téměř identické řešení jako INKS. Je nutno ovšem říci, že doba odezvy programu Multisim je výrazně rychlejší (neznatelná, na rozdíl od odhadovaných setin až desetin sekundy v našem případě).



Obrázek 6.3: Výsledek komerčního programu

7 Závěr

Úspěchem této práce bylo posílení slabého místa původní aplikace, kterým byla práce se spoji a z toho plynoucí diskomfort při obsluze programu. Práce splnila svůj cíl, kterým bylo zajistit automatické propojování spojů, jak v případě vytváření jednotlivého spoje, tak v případě vedení více spojů při posunu hradel. Podařilo se zajistit rozumnou dobu odezvy při těchto operacích, přestože výkon není tak vysoký jako u obdobných komerčních aplikací. Dále práce poskytla stručný souhrn algoritmů, které se v této oblasti dají využít, včetně vlastní modifikace algoritmu A*, která byla vytvořena do určité míry nezávisle.

Budoucí práce Možnou budoucí práci může představovat zrychlení výpočtu, pokud se ukáže, že stávající výkon není dostatečný. Rezervy ve výkonu lze hledat ve stávajícím zahazování hodnot v poli *Cost Buffer*, které by se v budoucnu mohlo udržovat i po provedení úprav ve schématu a použít pro propojení dalšího spoje. Časová náročnost by pak často klesla z $O(S^f \log S^f \cdot \log n)$ na $O(S^f \log S^f)$(viz 4.4).
Za úvahu také stojí zařazení dalších kritérií na tvar cesty, která by se promítla do tvaru cenové funkce. Jedná se například o vzdálenost od jiných vodičů.
Budoucí práce na projektu by měla zahrnovat doplnění některých dalších funkcí, které se od podobných programů očekávají. Zejména je to práce se spoji tvořenými více vodiči.

8 Literatura

[1] Rostislav Pastor: Interaktivní nástroj pro kreslení spojů, Bakalářská práce, FEL ČVUT v Praze, 2006

[2] Frank Rubin: The Lee Path Connection Algorithm, IEEE Computer Society, 1974

[3] Hai Zhou, EECS 357 Introduction to VLSI CAD, Detailed routing: shortest path and maze search, http://www.ece.northwestern.edu/~haizhou/357/lec6.pdf

[4] VLSI Algorithms, Global Routing, students.iiit.ac.in/maths/VLSI/Grouting.ppt

[5] Standard Template Library Programmer's Guide, Silicon Graphics, Inc., http://www.sgi.com/tech/stl/

[6] Dijkstra's Algorithm, Wikipedia, http://en.wikipedia.org/wiki/Dijkstra_algorithm

[7] A* search Algorithm, Wikipedia, http://en.wikipedia.org/wiki/A*

A Obsah přiloženého CD

Na přiloženém CD se nachází tyto adresáře:

- Kořenový adresář obsahuje soubor readme.txt
- *text* obsahuje bakalářskou práci v PDF a uživatelskou příručku
- *bin* obsahuje přeložený program
- *source* obsahuje zdrojové soubory