Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Anže Vrhovnik

Aproksimacijski algoritmi za reševanje problema najkrajšega hodnika

diplomsko delo
na univerzitetnem študiju

Mentor: doc. dr. Zoran Bosnić

Ljubljana, 2012

Univerza *v Ljubljani*
Fakulteta *za računalništvo in informatiko*

Tržaška 25
1001 Ljubljana, Slovenija
telefon: 01 476 84 11
01 476 83 87
faks: 01 426 46 47
01 476 87 11
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 01845/2012
Datum: 04.04.2012

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Kandidat: **ANŽE VRHOVNIK**

Naslov: **APROKSIMACIJSKI ALGORITMI ZA REŠEVANJE PROBLEMA NAJKRAJŠEGA HODNIKA**

APPROXIMATION ALGORITHMS FOR SOLVING THE MINIMAL CORRIDOR PROBLEM

Vrsta naloge: Diplomsko delo univerzitetnega študija

Tematika naloge:

Problem najkrajšega hodnika (angl. minimal corridor problem) je aktualen problem, saj z njim modeliramo hišne in druge napeljave (električne, vodovodne ipd.), ki se morajo razprostirati po vseh komponentah tlorisa. Iskanje optimalne rešitve je računsko zahteven in NP-poln problem.

Naloga kandidata bo predlagati različne hevristične algoritme za iskanje aproksimacijskih rešitev problema. Delovanje metode bo medseboj primerjal po času delovanja in kakovosti najdenih rešitev (glede na optimalno), za evalvacijo pa bo izbral množico umetno generiranih tlorisov.

Mentor:

doc. dr. Zoran Bosnić

Dekan:

prof. dr. Nikolaj Zimic

Fakulteta za računalništvo in informatiko • UNIVERZA V LJUBLJANI •

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Anže Vrhovnik, z vpisno številko **63050134**, sem avtor diplomskega dela z naslovom:

Aproksimacijski algoritmi za reševanje problema najkrajšega hodnika

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom doc. dr. Zorana Bosnića,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter besede (slov., angl.) identični s tiskano obliko diplomskega dela
- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki "Dela FRI".

V Ljubljani, dne 14. septembra 2012 Podpis avtorja:

Zahvaljujem se mentorju doc. dr. Zoranu Bosniću za predanost in pomoč pri izdelavi diplomske naloge ter za napotke, ki mi jih je nudil, ko sem zašel v slepo ulico.

Hvala tudi Ireni in vsem prijateljem za potrpežljivost, razumevanje in podporo.

Posebna zahvala gre tudi staršema, ki sta mi omogočila študij, me pri tem spodbujala in mi bila v oporo.

Kazalo

Povzetek

V diplomski nalogi bomo obravnavali problem minimizacije dolžine hodnika. Podan imamo tloris pravokotnega prostora, ki je sestavljen iz več sob. Poiskati moramo najkrajše možno omrežje, ki ima stik z vsako izmed sob in z zunanjim pravokotnikom.

V diplomski nalogi smo razvili tri aproksimacijske algoritme za reševanje tega problema in jih primerjali z algoritmom, ki z izčrpnim preiskovanjem poišče optimalno rešitev. Pri prvem od teh algoritmov skušamo zgraditi hodnik samo iz najkrajših sten. Drugi algoritem v hodnik sproti dodaja najboljše stene, dokler ne pridemo do dopustne rešitve. Pri tretjem algoritmu pa ocenjujemo vozlišča in jih dodajamo v hodnik glede na oceno, dokler ne pridemo do dopustne rešitve.

Izkazalo se je, da z uporabo aproksimacijskih algoritmov močno skrajšamo čas iskanja, vendar zaradi tega dobimo nekoliko slabše rezultate. Uporaba aproksimacijskih algoritmov je smiselna pri večjih problemih, saj bi na izračun optimalnih hodnikov čakali predolgo.

Ključne besede: minimizacija dolžine hodnika, aproksimacijski algoritem, najkrajša pot v grafu

Abstract

In the thesis we will discuss the minimum-length corridor problem. Given a rectangular boundary partitioned into rectangles, the problem is to find a minimum edge-length corridor that visits at least one point from the boundary of every rectangle and from the rectangular boundary.

We develop three approximation algorithms to solve this problem and compare them to the algorithm which always finds the optimal corridor. With the first algorithm, we try to build a corridor with the shortest walls only. The second algorithm adds the best walls to the corridor, until we get the acceptable solution. With the third algorithm, we value the vertexes and add the best of them to the corridor, until we achieve the acceptable result.

The results show that the use of approximation algorithms considerably shortens the search time but gives us slightly worse results. Using approximation algorithms is recommended for larger problems since the waiting time for calculation of optimum corridor would take too long.

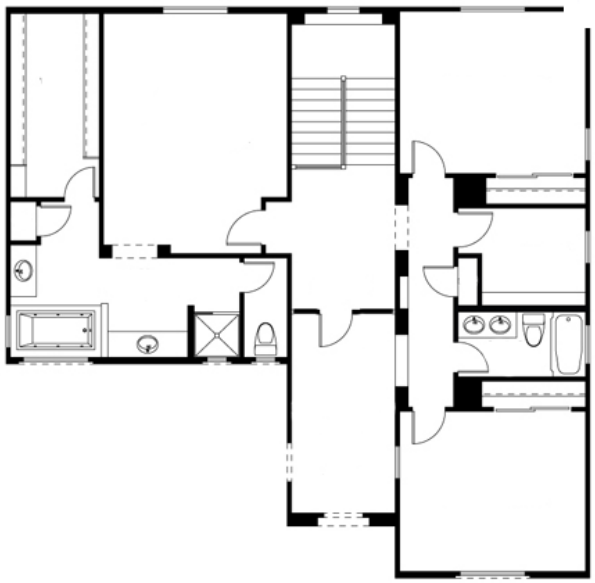
Keywords: minimum-length corridor problem, approximation algorithm, shortest path in a graph

Poglavje 1

Uvod

Mnogo ljudi se v svojem življenju odloči za gradnjo lastne hiše. Za marsikoga to predstavlja velik finančni zalogaj, saj običajno potrebujemo veliko delavcev in gradbenega materiala. Zato je smiselno preveriti, kje lahko kaj privarčujemo. Ena izmed možnosti je ustrezna napeljava različnih hišnih omrežij (računalniškega, električnega, vodovodnega, ...). Napeljati jo želimo tako, da bomo pri tem porabili čim manj gradbenega materiala (električnih vodnikov, cevi, ...). Primer razporeditve prostorov je prikazan na sliki 1.1. Problem v teoriji grafov poznamo kot problem minimizacije dolžine hodnika (angl. *minimal-length corridor problem, MLC*). Z njim se srečujemo tudi, ko v mestih napeljujemo razna omrežja (telefonsko, električno, vodovodno, optično). Vsaka stavba mora imeti dostop do omrežja, električne vodnike oziroma cevi pa lahko napeljujemo samo po ulicah mesta. Stavbe predstavljajo lica grafa, ulice pa povezave med lici grafa. Podoben primer najdemo tudi v elektrotehniki pri gradnji električnih vezij. Ko razporejamo električne elemente na silicijevo ploščico jih želimo razporediti čim bolj skupaj, da privarčujemo pri materialu. Problem lahko hitro prevedemo na problem minimizacije dolžine hodnika, kjer vsak električni element potrebuje vir energije in v grafu predstavlja sobe, povezave med elementi pa predstavljajo povezave v grafu.

V zunanji pravokotnik imamo vrisan ravninski graf, kjer vzporedne vo-



Slika 1.1: Primer razporeditve prostorov.

doravne in navpične povezave (stene) predstavljajo mejnike med različnimi pravokotnimi lici grafa (sobami). Na podmnožici točk grafa moramo poiskati takšno vpeto drevo, da se točke drevesa stikajo z vsemi lici grafa in je skupna dolžina povezav najmanjša. Prav tako se mora vsaj ena točka vpetega drevesa stikati z zunanjim pravokotnikom.

Izkaže se, da problem spada v množico računsko neobvladljivih problemov, ki so znani kot NP-polni problemi. To pomeni, da bi za rešitev poljubno velikega problema potrebovali zelo veliko časa kljub temu, da se je procesorska moč v zadnjih letih močno povečala. S tem se pozornost preusmeri na iskanje algoritmov, ki bi nam v čim krajšem času vrnili rešitev, ki bi bila čim bližje optimalni. Te algoritme imenujemo aproksimacijski algoritmi [13].

V diplomski nalogi bomo pregledali dosedanje raziskave na področju minimizacije dolžine hodnika in pogledali nekatere že znane aproksimacijske algoritme. Prav tako bomo predstavili nekatere algoritme, ki smo jih razvili sami in jih tudi testirali.

Poglavje 2

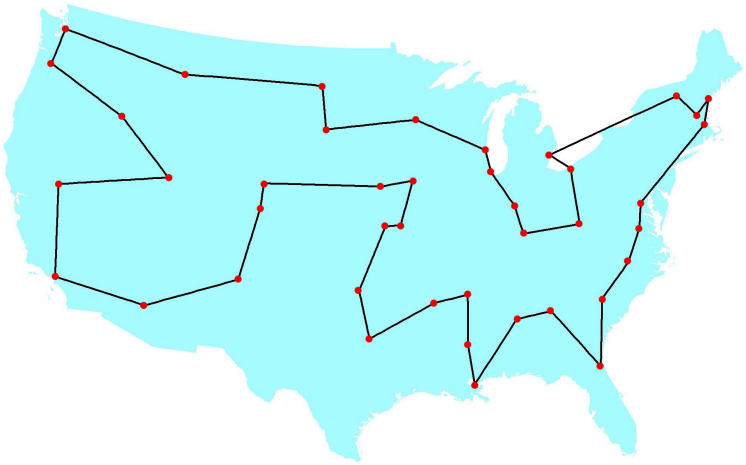
Opis problema

2.1 Izvor problema

Problem minimizacije dolžine hodnika spada med probleme, kjer imamo podano množico objektov v ravnini in moramo poiskati najkrajšo pot, ki se dotika vsakega izmed objektov vsaj enkrat. Pri tem moramo upoštevati določene omejitve. Najbolj znan soroden problem je problem trgovskega potnika, kjer imamo podano množico mest in ceno potovanja med poljubnima mestoma. Cena potovanja iz mesta X v mesto Y je enaka ceni potovanja iz mesta Y v mesto X. Poiskati moramo najcenejšo pot, kjer obiščemo vsa mesta in se na koncu vrnemo v izhodišče (primer na sliki 2.1).

Problem trgovskega potnika je eden izmed najbolj proučevanih problemov v računalništvu in zaenkrat še ne obstaja učinkovita rešitev, ki bi veljala v splošnem [14], [15]. Leta 1972 je Richard M. Karp dokazal, da je problem NP-poln. To je potrdilo računsko zahtevnost reševanja problema. Ena izmed bolj znanih rešitev je prav gotovo povezava 24978 mest na Švedskem. Problem je znan kot švedska pot (angl. *Sweden Tour*). Do leta 2006 največji rešen problem obsega povezavo 85900 mest. Gre za povezavo vozlišč na mikročipu.

Pri iskanju optimalne rešitve najprej pomislimo na preizkušanje vseh permutacij in nato pogledamo, katera izmed njih je najugodnejša. To nas privede do časovne zahtevnosti O(n!), kar pa je zelo težko obvladljivo že pri majh-



Slika 2.1: Rešitev problema trgovskega potnika za nekatera mesta v ZDA.

nih množicah mest. Ena izmed možnosti je Held-Karpov algoritem, ki nas privede do rešitve v času $O(n^2 2^n)$. Z uporabo dinamičnega programiranja je možno problem rešiti celo v času $O(2^n)$. Iskanje učinkovitejšega algoritma je zelo težko. Do sedaj še ni znano, če obstaja algoritem, ki nam poišče optimalno pot v času $O(1.9999^n)$.

Problem trgovskega potnika se v praksi lahko uporablja še drugje kot samo pri logistiki in načrtovanju potovanj. Z njim se pogosto srečamo pri načrtovanju mikročipov. Če ga nekoliko priredimo, ga lahko uporabimo tudi v genetiki pri raziskavi DNA.

2.2 Problem minimizacije dolžine hodnika

Problem minimizacije dolžine hodnika je podoben pred tem omenjenemu problemu trgovskega potnika. Namesto mest imamo tu sobe, ki jih lahko obiščemo na več mestih (vozliščih). Povezave med mesti nadomestimo s stenami sobe. Ena izmed razlik je tudi v tem, da nam pri problemu minimizacije dolžine hodnika ni treba poiskati krožne poti, ampak moramo samo obiskati vse sobe. Poleg tega ima pot lahko tudi odcepe.

2.2.1 Zgodovina problema

Problem minimizacije dolžine hodnika (MLC) je prvi predstavil Naoki Katoh leta 2001 na 13th Canadian Conference of Computional Geometry [4]. Nato je leta 2002 David Eppstein predstavil različico z omejitvijo, da morajo vse sobe imeti obliko pravokotnika (MLC-R) [5]. Zdelo se je, da je tako problem minimizacije dolžine hodnika kot tudi problem minimizacije dolžine hodnika s pravokotnimi sobami NP-poln, vendar za to ni bilo nobenega dokaza. Znan ni bil niti noben aproksimacijski algoritem s konstantnim aproksimacijskim faktorjem. Aproksimacijski faktor nam pove, največ kolikokrat je rešitev aproksimacijskega algoritma slabša od optimalne. Če imamo algoritem z aproksimacijskim faktorjem f in je optimalna dolžina hodnika za primer i enaka n_i, to pomeni, da je dolžina hodnika, ki ga dobimo z aproksimacijskim algoritmom, manjša ali enaka $f \cdot n_i$. Torej ne obstaja primer, kjer bi nam aproksimacijski algoritem vrnil hodnik, ki bi imel dolžino več kot $f - krat$ daljšo od optimalne.

Konec leta 2006 so Hans L. Bodlaender in ostali [3] ter neodvisno od njega še skupaj Arturo Gonzalez-Gutierrez ter Teofilo F. Gonzalez [7] dokazali, da sta tako problem MLC kot tudi MLC-R strogo NP-polna. S tem se je odprla možnost iskanja aproksimacijskih algoritmov, ki bi nam v polinomičnem času vrnili rešitev, ki bi bila čim bližje optimalni.

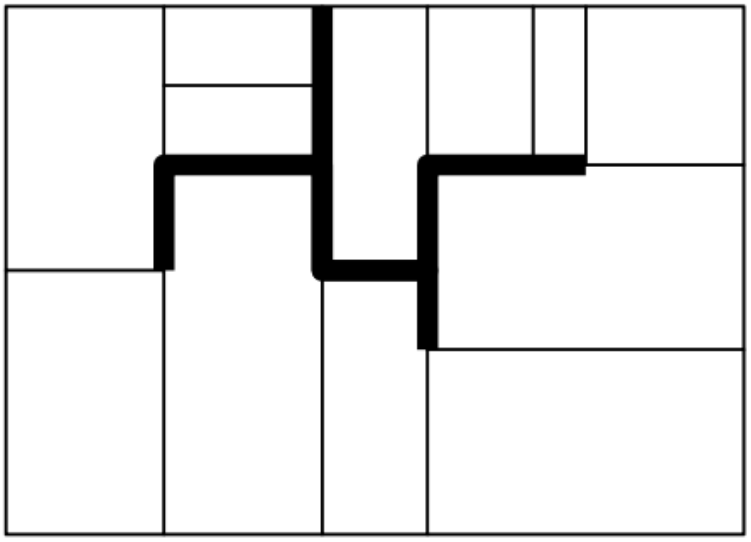
Prav tako so leta 2006 Hans L. Bodlaender in ostali [3] predstavili nekatere aproksimacijske algoritme, ki bodo predstavljeni v nadaljevanju. Nobeden izmed njih nima konstantnega aproksimacijskega faktorja.

Leta 2007 je Arturo Gonzalez-Gutierrez predstavil aproksimacijski algoritem z aproksimacijskim faktorjem 30. To je bil prvi algoritem za reševanje tega problema z dokazljivo konstantnim aproksimacijskim faktorjem.

2.2.2 Definicija problema

V tej diplomski nalogi se bomo posvetili problemu minimizacije dolžine hodnika v ravnini s pravokotnimi sobami (MLC-R).

Podan imamo zunanji pravokotnik F, ki ima več vpetih pravokotnikov (ali sob) P_1, P_2,...,P_n. Poiskati moramo množico daljic S tako, da vsaka daljica leži na stranici katere izmed sob ali na stranici zunanjega pravokotnika F. Daljice v množici S tvorijo drevo in vključujejo vsaj eno točko iz katerekoli stranice vsake sobe in vsaj eno točko iz katerekoli stranice zunanjega pravokotnika F. Vsota dolžin daljic v množici S se imenuje dolžina hodnika in jo označimo z $L(S)$. Cilj je poiskati takšno množico S, ki bo ustrezala zgornjim pogojem in bo imela najmanjšo dolžino hodnika $L(S)$ [6]. Na sliki 2.2 je najkrajši hodnik prikazan z odebeljeno črno črto.



Slika 2.2: Problem najmanjše dolžine hodnika.

Problem si lahko predstavljamo kot tloris hiše, ki je razdeljena na več pravokotnih sob. Po stenah moramo napeljati napeljavo tako, da ima ta stik z vsako izmed sob in z zunanjim svetom. Med vsemi možnimi rešitvami moramo poiskati najkrajšo.

2.2.3 Različice problema

Poleg problema MLC-R obstaja na tem področju tudi več podobnih problemov, od katerih ima vsak svoje značilnosti:

- **MLC** (angl. *Minimum-Length Corridor*) je najbolj splošna različica problema. Razlika v primerjavi z MLC-R je v tem, da MLC ne zahteva, da so sobe pravokotniki. Dovolj je, če so stene sob vzporedne s koordinatnima osema.

- **$\mathbf{MLC_k}$** je strožja različica problema MLC, kjer je vsaka soba c-kotnik za vsak $c \leq k$.

- **p-MLC-R** (angl. *p-access point MLC-R*) je različica problema MLC-R, kjer mora hodnik obvezno vsebovati vstopno točko p, ki leži na zunanjem pravokotniku.

- **TRA-MLC** (angl. *top-right access point MLC*) je različica problema MLC, kjer mora hodnik obvezno vsebovati zgornji desni kot zunanjega pravokotnika kot vstopno točko. Problem je podrobneje obravnaval Arturo Gonzalez-Gutierrez v [6].

- **N-MLC** (angl. *network MLC*) je posplošitev problema MLC na grafe. Podan imamo povezan neusmerjen uravnotežen graf. Poiskati moramo drevo z najmanjšo skupno ceno tako, da ima vsak cikel v grafu vsaj eno izmed svojih vozlišč v drevesu.

Poglavje 3

Pregled izbranih aproksimacijskih algoritmov

V tem poglavju bomo na kratko predstavili nekatere znane aproksimacijske algoritme. Najprej bomo predstavili delo Hansa L. Bodlaenderja in ostalih [3], nato pa še prispevek Artura Gonzaleza-Gutierezza ter Teofila F. Gonzaleza [6]. Algoritmi so namenjeni različnim variantam problema minimizacije dolžine hodnika.

3.1 Aproksimacijski algoritem za problem MLC s sobami različnih velikosti

Hans L. Bodlaender in ostali [3] so predstavili aproksimacijski algoritem za problem MLC s sobami različnih velikosti (angl. *An approximation algorithm for MCC with rooms of varying sizes*). Avtorji so v angleščini problem poimenovali MCC (angl. *minimum corridor connection*). Pravzaprav je problem enak MLC, samo poimenovanje je drugačno.

Pri tem algoritmu je aproksimacijski faktor odvisen od *debeline* sob. Za vsako sobo $R_i, i \in [1, ..., n]$ določimo njeno *debelino* ρ_i, ki predstavlja dolžino stranice najmanjšega kvadrata, ki jo lahko obdaja. Obseg vsake je omejen navzgor s $4\rho_i$. Sobi R_i rečemo, da je $\alpha - debela$ (angl. $\alpha - fat$), če za ka-

terikoli kvadrat Q, katerega stranice sekajo sobo R_i in njegovo središče leži v R_i, velja, da je presek površin kvadrata Q in sobe R_i najmanj $\alpha/4 - krat$ površina kvadrata Q. V splošnem velja $\alpha \in [0..1]$. Debelina kvadrata je 1.

Algoritem GREEDY:

1. Izberemo oglišča $p_i \in R_i, i \in \{1, ..., n\}$ tako da minimiziramo $\sum_{i=2}^{n} d(p_1, p_i)$, kjer je $d(x, y)$ najkrajši hodnik med x in y.

2. Naj bo G graf z vozliščem v_i za vsako sobo R_i in $d(v_i, v_j) = d(p_i, p_j)$. Poiščemo najmanjše vpeto drevo T in G.

3. Za vsako oglišče (v_i, v_j) v grafu T naj najkrajša pot (p_i, p_j) pripada hodniku. Če novonastali hodnik ni drevo, potem odstranimo cikle (odstranimo oglišča).

Algoritem GREEDY nam vrne rešitev z aproksimacijskim faktorjem $(n - 1)$. Prav tako je dolžina najkrajšega hodnika, ki povezuje k sob, najmanj $\rho_{min}(k\alpha/2 - 2)$, kjer je ρ_{min} velikost najmanjše izmed teh sob.

Algoritem CONNECT:

1. Sobe razporedimo po velikosti $\rho_1 \leq \rho_2 \leq ... \leq \rho_n$. Izberemo poljuben p_1, ki leži na stranici sobe R_1. Za i=2 do n izberemo točko p_i na stranici sobe R_i tako, da minimiziramo $min\{d(p_i, p_1), d(p_i, p_2), ..., d(p_i, p_{i-1})\}$

2. Naj bo G graf z vozliščem v_i za vsako sobo R_i in $d(v_i, v_j) = d(p_i, p_j)$. Poiščemo najmanjše vpeto drevo T in G.

3. Za vsako oglišče (v_i, v_j) v grafu T naj najkrajša (p_i, p_j) pot pripada hodniku. Če novonastali hodnik ni drevo, potem odstranimo cikle (odstranimo oglišča). Za rešitev vzamemo minimum med tako dobljenim drevesom in drevesom, dobljenim z algoritmom GREEDY.

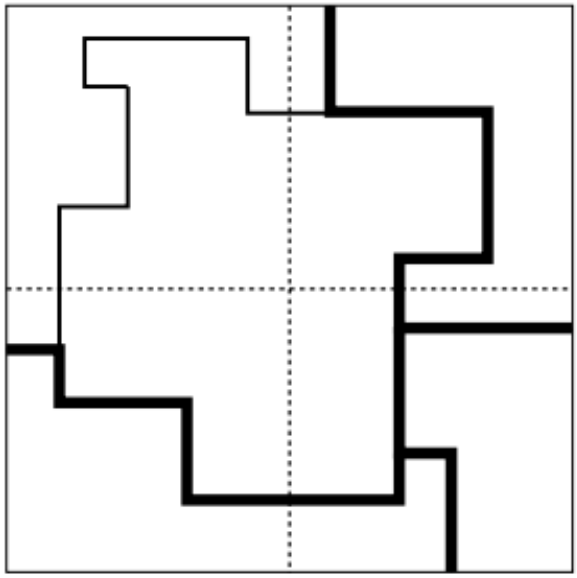
Algoritem CONNECT nam vrne rešitev z aproksimacijskim faktorjem $(16/\alpha - 1)$, kjer je debelina vsake sobe najmanj α.

3.2 Aproksimacijski algoritem z geografskim združevanjem

Aproksimacijski algoritem za problem MLC z geografskim združevanjem (angl. *A PTAS for MCC with geographic clustering*) je algoritem, kjer za izhodišče vzamemo algoritem za reševanje problema trgovskega potnika, ki ga je obravnaval Arora [2]. Predpostavimo, da so koordinate vsakega oglišča vsake izmed n sob elementi množice celih števil, da lahko kvadrat velikosti $q \cdot q$ včrtamo v vsako sobo in da ima vsaka soba obseg največ $c \cdot q$ za neko konstanto $c \geq 4$. Definiramo tudi mejni kvadrat (angl. *bounding box*), ki je najmanjši možni kvadrat, ki vsebuje vse sobe. Njegove stranice ležijo vzporedno s koordinatnima osema in njegova stranica je dolga največ cqn. Velikost mejnega kvadrata označimo s črko $L \in [cqn, 2cqn]$.

Najprej definirajmo ravno delitev (angl. *straight dissection*) mejnega kvadrata. Mejni kvadrat razdelimo na štiri enako velike kvadrate. Vsakega izmed teh kvadratov nato zopet razdelimo na štiri enako velike kvadrate. Delitev zaključimo, ko je dolžina stranice kvadrata enaka cq. Ker je $L \leq 2cqn$ ima ta operacija časovno zahtevnost $O(logn)$. Nivo kvadrata je globina kvadrata v delitvenem drevesu in delitvene črte nivoja i so ravne črte, ki delijo kvadrat na nivoju $i-1$ v kvadrate na nivoju i. Delitvena črta lahko razdeli sobo v dva ali več delov. To predstavlja pri dinamičnem programiranju težavo, ker moramo določiti, katera soba pripada kateremu kvadratu. Da rešimo ta problem, definiramo ukrivljeno delitev (angl. *curved dissection*).

Zamislimo si vodoravno delitveno črto. Zamenjamo jo z delitveno krivuljo, ki nastane tako, da gremo z leve proti desni in ko pridemo do stranice katere izmed sob, sledimo stranici (v poljubni smeri) dokler zopet ne pridemo do delitvene črte. Krivuljo skrajšamo in ustvarimo pot, ki razdeli množico sob v zgornjo in spodnjo. Navpične delitvene črte so definirane podobno. Postopek lahko izvedemo tako, da vsaka vodoravna krivulja prečka vsako navpično krivuljo natanko enkrat. Vodoravne oziroma navpične krivulje se ne sekajo med seboj. Pretvorba delitvenih črt v delitvene krivulje preslika



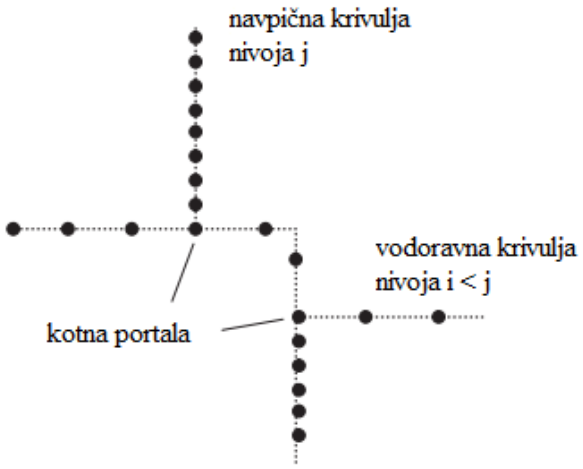
Slika 3.1: Ravna (prekinjena črta) in ukrivljena (neprekinjena odebeljena črta) delitev.

vsako vozlišče ravnega delitvenega drevesa v mnogokotnik, ki ga označimo kot *vozliščni mnogokotnik*.

Na sliki 3.1 so delitvene črte prikazane s tanko prekinjeno črto, delitvene krivulje pa z odebeljeno neprekinjeno črto. Kot lahko opazimo na sliki, srednjo sobo prečkata tako vodoravna kot navpična delitvena črta.

Na nivoju i naj ima delitvena krivulja $2^i m$ posebnih točk, ki so enakomerno razporejene po njej. Te točke bomo imenovali *portali*, število m pa *parameter portala*. Vedeti moramo, da je presek vodoravne in navpične krivulje v splošnem pot. Po teh poteh torej potekata dve množici portalov. Eden od vodoravne krivulje, drugi od navpične krivulje. Na teh delih obdržimo samo množico portalov, ki ležijo na krivulji višjega nivoja. Če sta oba nivoja enaka, potem izberemo poljubno množico. Definiramo tudi po en portal na robu vsake izmed poti in jih imenujemo *kotni portali*. Kotna portala sta prikazana na sliki 3.2

Da bomo problem lahko rešili z uporabo dinamičnega programiranja, mo-



Slika 3.2: Kotna portala.

ramo predvideti, da če del drevesa sovpada z delitveno krivuljo, potem lahko povezuje samo sobe na eni strani krivulje. Da povežemo sobe na drugi strani krivulje, jo moramo prečkati. Če se prehodi dogajajo samo v portalih, potem tako drevo imenujemo portalno drevo (angl. *portal respecting tree*). Stranice vozliščnega mnogokotnika, ki ga tvori delitvena krivulja, bomo poimenovali rob vozliščnega mnogokotnika. Robovi se lahko tudi prekrivajo. Portalno drevo je $k - lahko$, če prečka vsak rob vsakega vozlišča sobe največ $k - krat$

Algoritem poteka tako, da najprej naredimo mejni kvadrat z delitvenimi krivuljami. Zatem izberemo števili $a, b \in \{1, 2, ..., L/(cq)\}$ in vodoravna a in navpična b delitvena krivulja postaneta krivulji na nivoju 0. Sedaj zgradimo delitveno drevo tako, kot je to opisano v [1]. Zatem definiramo portale. Nato začnemo pri listih delitvenega drevesa in od spodaj navzgor posodobimo tabelo dinamičnega programiranja. Za vsak vozliščni mnogokotnik, ki ga tvori delitvena krivulja, za vsako podmnožico k portalov na stranicah sobe in za vsako kombinacijo $B1, ..., B_p$ teh k portalov shranimo dolžino optimalnega gozda, ki je sestavljen iz p dreves, ki se skupaj dotikajo vseh sob in $i-to$ drevo povezuje vse portale v B_i. Za vozliščne mnogokotnike v listih delitvenega drevesa navedemo vse take gozdove. Vsak izmed njih vsebuje največ c^2 sob.

Za korenski vozliščni mnogokotnik pridobimo podatek o portalih iz dveh delitvenih krivulj na nivoju 1, ki razdeljujeta korenski vozliščni mnogokotnik. Zagotoviti moramo, da štirje gozdovi skupaj tvorijo eno drevo.

3.3 Algoritem s konstantnim aproksimacijskim faktorjem

Arturo Gonzalez-Gutierrez je predstavil nekatere algoritme za iskanje najkrajše dolžine hodnika za problem MLC-R [6]. V splošnem je algoritem definiran kot Alg(S), kjer je S izbirna funkcija. Ideja algoritma je, da v vsakem pravokotniku omejimo število možnih točk, od katerih mora biti vsaj ena vključena v hodnik. Te točke bomo imenovali kritične točke. Izbirna funkcija S jih izbira enakomerno. S $k_s \geq 1$ bomo označili največje število kritičnih točk, ki jih funkcija S izbere v vsaki sobi R_i. Problem se imenuje MLC-R_S. Cilj je poiskati tak najkrajši hodnik, ki vsebuje vsaj eno kritično točko iz vsakega pravokotnika (sobe).

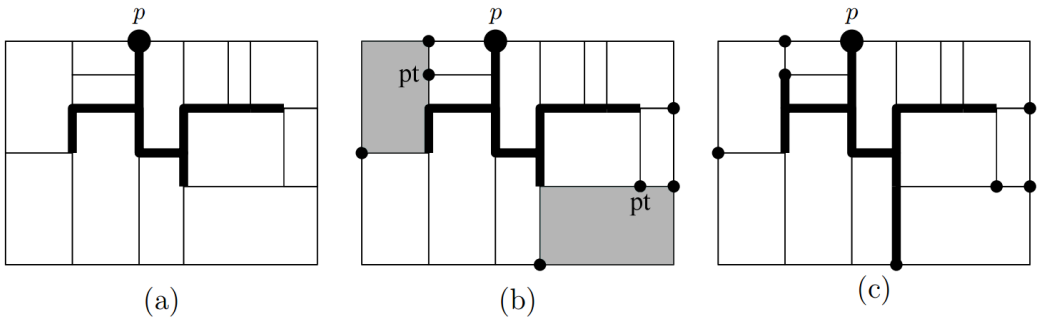
V nadaljevanju bomo predstavili prvi aproksimacijski algoritem za reševanje problema minimizacije dolžine hodnika s konstantnim aproksimacijskim faktorjem.

3.3.1 Izbirne funkcije

Definirali bomo tri izbirne funkcije in sicer S($4C$), S($F4C$) in S(R_k). Izbirna funkcija S($4C$) v vsaki sobi izbere vsa štiri oglišča, funkcija S($F4C$) izbere v vsaki sobi manj kot 4 oglišča in S(R_k) v vsaki sobe izbere k naključnih točk. Nobena izmed teh izbirnih funkcij nas ne privede do algoritma s konstantnim aproksimacijskim faktorjem. Do njega lahko pridemo tako, da v vsaki sobi poljubno izberemo dva nasprotna kota in eno posebno točko. Posebna točka je vozlišče sobe R_i, ki ni eno izmed njenih oglišč, in je razdalja, ki je potrebna za povezavo tega vozlišča z vsemi ostalimi delnimi hodniki v množici A, najmanjša možna. V množici A so vsi delni hodniki, ki ne vključujejo vozlišča

iz sobe R_i, vključujejo pa vozlišča iz vseh preostalih sob. Iskanje posebnih točk je v tem primeru časovno zahtevno. Če je množica delnih hodnikov prazna, posebnih točk ne moremo določiti. V teh primerih lahko uporabimo zgornjo mejo, ki je potrebna za povezavo vozlišča z vsemi ostalimi delnimi hodniki.

Primer gradnje hodnika z aproksimacijskim faktorjem je prikazan na sliki 3.3. Brez izgube splošnosti bomo privzeli, da za izbiro nasprotnih kotov vedno vzamemo zgornji desni in spodnji levi kot. Na sliki 3.3(a) je prikazana optimalna rešitev problema. Vsak izmed pravokotnikov je povezan v hodnik skozi vsaj eno izmed svojih kritičnih točk razen dveh, ki sta na sliki 3.3(2) obarvana s sivo barvo. Posebni točki sta na teh dveh pravokotnikih označeni s *pt*. Če želimo iz sivih pravokotnikov v hodnik vključiti še kritične točke iz teh dveh pravokotnikov, potem dobimo rešitev, ki je prikazana na sliki 3.3(c).



Slika 3.3: Gradnja hodnika s konstantnim aproksimacijskim faktorjem.

Poglavje 4

Uporabljene tehnologije

Algoritme, predstavljene v nadaljevanju, smo razvili v jeziku C#, ki teče na ogrodju .NET Framework 3.5. Pri pisanju smo uporabili razvojno okolje Microsoft Visual Studio 2008 Proffesional edition.

4.1 C#

C# je objektno orientiran programski jezik, ki ga je razvilo podjetje Microsoft [10]. Je naslednik jezika C++ in je zelo razširjen. Razvijalcem omogoča izdelovanje običajnih aplikacij za operacijski sistem Windows, spletnih storitev, aplikacij odjemalec - strežnik in še mnogo več. Jezik je berljiv in prijazen do razvijalcev.

4.2 .NET Framework 3.5

Programi, napisani v programskem jeziku C#, tečejo na ogrodju .NET Framework [11]. Ogrodje je razvilo podjetje Microsoft in teče primarno na operacijskih sistemih Microsoft Windows. Vključuje navidezni izvajalni sistem (angl. *common language runtime - CLR*) in množico knjižnic, ki vključujejo uporabniški vmesnik, dostop do podatkov, povezovanje z bazo podatkov, kriptografijo, razvoj spletnih aplikacij, številčne algoritme in mrežne komu-

nikacije. Razvijalci kombinirajo njihovo lastno kodo z ogrodjem .Net Framework in drugimi knjižnicami.

4.3 Visual Studio 2008

Visual Studio 2008 je razvojno okolje, ki ga je prav tako razvilo podjetje Microsoft [12]. Podpira razvijanje aplikacij v programskih jezikih C/C++, VB.NET, C#,F#. Podprt je tudi razvoj v drugih jezikih kot so M, Python, Ruby, XML/XSLT, HTML, JavaScript in CSS. Vsebuje urejevalnik kode s podporo za IntelliSense in preoblikovanje (angl. *refactoring*) kode ter razhroščevalnik (angl. *debugger*). Na voljo so orodja za oblikovanje zaslonskih mask, spletni urejevalnik (angl. *web designer*), urejevalnik razredov (angl. *class designer*) in urejevalnik sheme podatkov.

Poglavje 5

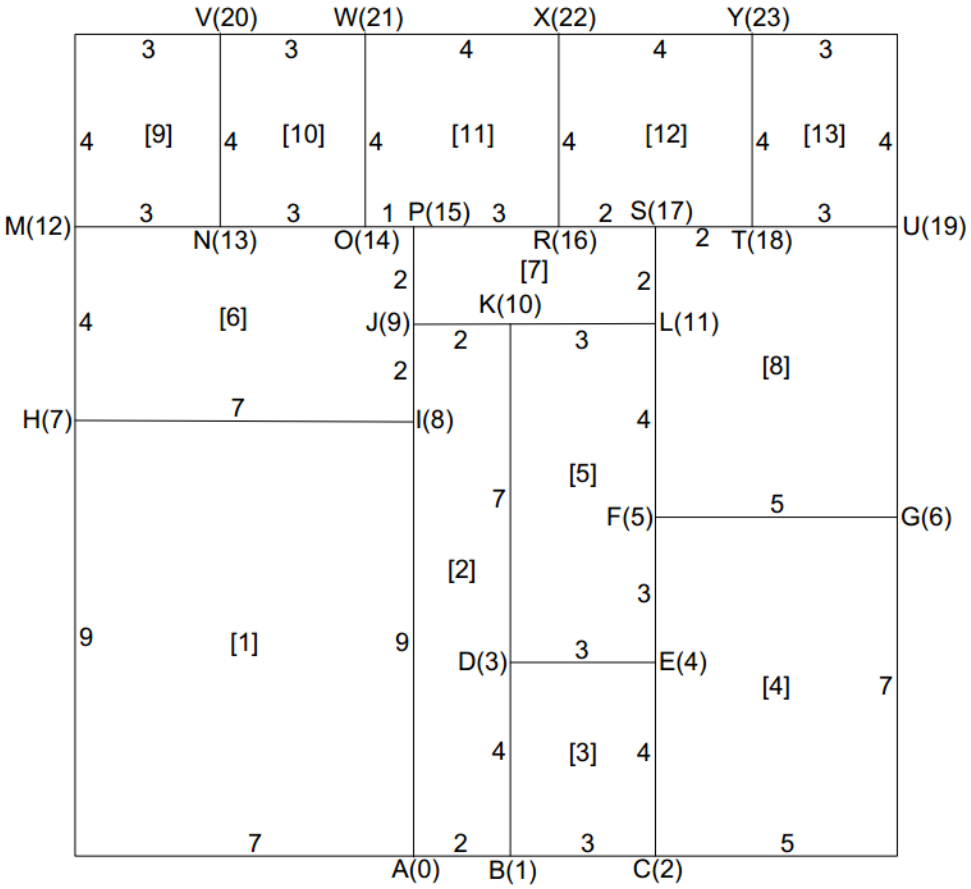
Izčrpno preiskovanje rešitev najkrajšega hodnika

V nadaljevanju bomo predstavili algoritem za reševanje problema najkrajšega hodnika s pravokotnimi sobami, ki nam vedno vrne optimalno rešitev problema. Najprej bomo prikazali kako smo skico problema pretvorili v računalniku razumljivo obliko, sledi nekaj napotkov za programiranje in sam algoritem.

5.1 Pretvorba skice problema v računalniško obliko

Za delovanje algoritma potrebujemo ustrezne podatke o sobah, stenah in vozliščih. Zato smo razvili postopek, kako označiti vse bistvene elemente problema in jih nato zapisati v tabele.

Denimo, da moramo rešiti problem, ki je prikazan na sliki 5.1. Kot vidimo je pravokotnik razdeljen na 13 sob, ki so sestavljene iz 40 sten. Vozlišča so označena z velikimi tiskanimi črkami poleg katerih se nahaja tudi njihov indeks, ki ga bomo uporabili v nadaljevanju. Ob stenah so zapisane številke. Vsaka predstavlja dolžino stene v enotah mere. V vsaki sobi se nahaja tudi številka v oglatih oklepajih. Ta predstavlja zaporedno številko sobe. Vrstni red poimenovanja ni pomemben.



Slika 5.1: Testni primer

Sedaj moramo skico problema pretvoriti v računalniško obliko. Odločili smo se, da bomo uporabili dve tabeli. V eno bomo zapisali podatke o tem, na katere stene meji vsako izmed vozlišč. V drugo bomo shranili podatke o stenah. Sobe smo zapisali v dvodimenzionalno tabelo *sobe* velikosti *stVozlisc* x 4, kjer je *stVozlisc* število vseh vozlišč. Indeks prve dimenzije predstavlja indeks vozlišča, indeks druge dimenzije pa teče od 0 do 3, saj lahko vsako vozlišče meji na največ štiri sobe. Vrednost v tabeli na indeksu [i,j] predstavlja j-to sobo vozlišča i. Vozlišče D z indeksom 3 meji na sobe [2], [3] in [5], zapis pa izgleda tako:

```
sobe[3, 0] = 2;
sobe[3, 1] = 3;
sobe[3, 2] = 5;
```

Stene predstavimo nekoliko drugače. Predstavljene so z dvodimenzionalno tabelo *stene* velikosti *stSten* x 3, kjer je *stSten* število vseh sten. Prva koordinata predstavlja zaporedno številko stene. Številčenje sten je lahko poljubno. Druga koordinata je tabela velikosti 3, ki na indeksu 0 hrani indeks začetnega vozlišča, na indeksu 1 je indeks končnega vozlišča, na indeksu 2 pa dolžina stene. Stena iz vozlišča D(3) v vozlišče K(10) je shranjena tako:

```
stene[i] = new int[] {3,10,7}
```

kjer je i poljubno število med 0 in [stSten -1].

5.2 Ideja algoritma za iskanje najkrajšega hodnika in optimizacija

Algoritma smo se lotili tako, da smo vzeli vse možne kombinacije sten. Nato smo preverili, če je posamezna rešitev dopustna. Če je in ima hodnik manjšo dolžino, kot trenutno najugodnejša rešitev, potem ta rešitev postane trenutno najugodnejša. Postopek ponavljamo, dokler ne pregledamo vseh možnih kombinacij sten.

Če se iskanja rešitve lotimo sistematično in iščemo rešitve, kjer je število sten enako $n \in \{1, 2, ..., stSten\}$, ugotovimo, da se prva dopustna rešitev pojavi pri nekem n. Če bi torej poznali, kateri je ta n, bi bilo iskanje najugodnejše rešitve hitrejše. V znani literaturi nismo nikjer zasledili podatka o tem, kako bi prišli do najmanjšega števila sten, ki morajo biti vključene v hodnik, da se dokopljemo do dopustne rešitve. Lotili smo se iskanja te mejne vrednosti in odkrili formulo za izračun števila sten, ki morajo biti vključene v hodnik, da zagotovo pridemo do dopustne rešitve. Ta formula se glasi:

$$maxSten = n_{st} \text{ - } n_{so} \text{ - } \lceil n_{vo}/2 \rceil$$

kjer so:

$maxSten$ - število sten, ki so vključene v hodnik, kjer se zagotovo pojavi rešitev.

n_{st} - število vseh sten

n_{so} - število vseh sob

n_{vo} - število vseh vozlišč

Pri štetju števila sten štejemo steni, ki se stikata v oglišču zunanjega pravokotnika, kot eno steno. Njena dolžina je enaka vsoti dolžin obeh sten. Prav tako oglišča zunanjega pravokotnika ne spadajo med vozlišča.

To formulo uporabimo za izhodišče algoritma, kjer bomo iskali rešitve. Ko bomo našli rešitev problema za $maxSten$ sten, potem bomo skušali najti rešitev za $maxSten - 1$ sten. Če takšna rešitev obstaja, potem bomo skušali najti rešitev za $maxSten - 2$ sten in tako dalje, dokler dobivamo dopustne rešitve. Ko enkrat dopustne rešitve ne moremo več najti, se algoritem konča.

Z uporabo te formule smo skrajšali čas iskanja rešitve, saj nam ni treba pregledovati nekaterih kombinacij sten, ki bi nam vrnile slabše rezultate ali kjer rešitev sploh ni možna. V praksi nam ni uspelo poiskati protiprimera, kjer zgornja formula ne bi držala.

5.3 Algoritem za minimizacijo dolžine hodnika

Pred začetkom izvajanja algoritma moramo pripraviti nekatere spremenljivke, ki jih bomo potem potrebovali pri iskanju rešitve. Nato poiščemo najboljšo rešitev in jo izpišemo v človeku razumljivi obliki.

5.3.1 Priprava na algoritem

Najprej skico primera pretvorimo v obliko, ki je razumljiva računalniku. Postopek smo opisali v poglavju 5.1. S tem postopkom napolnimo tabeli *sobe* in *stene*. Nastavimo tudi vrednost konstantam *stSten*, *stSob* in *stVozlisc*. V *stSten* imamo shranjeno število vseh sten. Stena je povezava iz enega vozlišča v drugo. *stSob* predstavlja število sob v grafu, *stVozlisc* pa število vseh vozlišč.

Definiramo tudi tabelo *vstopi*. V njej so shranjeni indeksi vseh vozlišč, ki ležijo na zunanjem pravokotniku. Vsaj eno vozlišče iz tabele *vstopi* mora biti vsebovano v končnem hodniku. Tabelo napolnimo tako, da ji dodamo vsa vozlišča, ki se dotikajo samo dveh sob. Ta vozlišča namreč ležijo na zunanjem pravokotniku.

Pred pričetkom algoritma sortiramo tabelo *stene* tako, da so krajše stene na začetku, daljše pa na koncu tabele. S tem bomo v večini primerov hitreje prišli do ugodne rešitve, saj bomo najprej preskusili algoritem na krajših stenah. Za veliko kombinacij nam ne bo treba preverjati dopustnosti rešitve, saj bo vsota dolžin njihovih sten daljša od trenutno najugodnejše rešitve.

Izračunati moramo še število sten, ki jih bomo vključili v hodnik. Za to uporabimo formulo 5.2. Vrednost shranimo v spremenljivko *maxSten*.

5.3.2 Potek algoritma

Vzamemo vse možne kombinacije sten, kjer je število sten natanko *maxSten*. Pri vsaki kombinaciji najprej preverimo vsoto dolžin njenih sten, ker je časovna zahtevnost te operacije samo $O(n)$. Velikokrat se nam lahko namreč primeri, da je vsota dolžin sten večja ali enaka od trenutno najkrajšega

hodnika in nam ni treba preverjati, če je rešitev dopustna ali ne. Če je torej vsota dolžin sten večja od trenutno najkrajšega hodnika, nadaljujemo preverjanje z naslednjo kombinacijo.

Če je vsota dolžin sten manjša od trenutno najkrajšega hodnika, potem preverimo dopustnost rešitve. Rešitev je dopustna, če zadošča naslednjim trem pogojem:

1. kombinacija sten vsebuje vsaj eno vozlišče, ki leži na zunanjem pravokotniku (hodnik ima stik z zunanjim pravokotnikom),

2. kombinacija sten mora imeti stik z vsemi sobami,

3. stene morajo biti povezane med seboj tako, da tvorijo hodnik (ne sme biti nepovezanih delov).

Preverjanje pogoja 1 opravimo tako, da po vrsti za vsako steno v kombinaciji preverimo, če je katero od njenih vozlišč v tabeli *vstopi*. Če najdemo takšno vozlišče, potem je pogoju zadoščeno.

Če je pogoju 1 zadoščeno, se lotimo preverjanje pogoja 2. Pri vsaki steni za vsako izmed njenih vozlišč preverimo, s katerimi sobami se stikata. Pri tem uporabimo tabelo *sobe*. Če se vozlišča stikajo z vsemi sobami, potem je rešitev dopustna.

Če smo prestali prva dva pogoja, se lotimo še pogoja 3. Najprej uvedemo tabeli *tempHodnik*, kamor prekopiramo kombinacijo sten, in tabelo *dodanaVozlisca*, kamor bomo shranjevali vozlišča, ki so v novonastajajočem hodniku. Sedaj vzamemo prvo steno iz *tempHodnik* in dodamo njeni vozlisci v tabelo *dodanaVozlisca*, steno pa izbrišemo iz *tempHodnik*. Nato za vsako steno preverimo, če je katero izmed njenih vozlišč že v tabeli *dodanaVozlisca*. Če je, potem v tabelo *dodanaVozlisca* dodamo preostalo vozlišče in steno izbrišemo iz tabele *tempHodnik*. Postopek ponavljamo, dokler v *tempHodnik* obstaja stena, ki ima katero izmed vozlišč enako kateremu izmed vozlišč v tabeli *dodanaVozlisca*. Po koncu postopka preverimo, če je tabela *tempHodnik* prazna. Če je, to pomeni, da so vse stene povezane med seboj in je pogoju

zadoščeno. Če tabela *tempHodnik* ni prazna, to pomeni, da nekatere stene niso povezane z ostalimi in rešitev ni dopustna.

Preverjanje dopustnosti rešitve je v nadaljevanju predstavljeno s psevdokodo.

```
bool imaVstop = false;
int[] s = new int[stSob + 1];
//preverimo, če hodnik ima vstopno točko
for (i = 0 to maxSten-1)
{
  if (vsebujeVstop(hodnik[i]) imaVstop = true;
}
if (!imaVstop)
  return false;
//preverimo, če se hodnik stika z vsemi sobami
for (i = 0 to maxSten-1)
{
  for (i = 0 to 3)
  {
    if(sobe[hodnik[i][0],j] != -1)
      s[sobe[hodnik[i][0],j]] = 1
    if (sobe[hodnik[i][1],j] != -1)
      s[sobe[hodnik[i][1],j]] = 1
  }
}
for (i = 0 to s.Length)
  if (s[i] == 0)
    return false;
//preverimo še, če so stene povezane med seboj
int[][] tempHodnik = new int[maxSten][];
Array.Copy(hodnik, tempHodnik, maxSten);
```

```
int[] dodanaVozlisca = new int[stVozlisc];
for (int i to dodanaVozlisca.Length; i++)
  dodanaVozlisca[i] = -1;
int stdodanaVozlisca = 0;
int dodanaVozliscaLast = 2;
dodanaVozlisca[0] = tempHodnik[0][0];
dodanaVozlisca[1] = tempHodnik[0][1];
tempHodnik[0] = null;
while (st < maxSten)
{
  for (int i = 0; i < maxSten; i++)
  {
    if (tempHodnik[i] == null) continue;
    if (tempHodnik[i][0] == dodanaVozlisca[st])
    {
      dodanaVozlisca[dodanaVozliscaLast] = tempHodnik[i][1];
      tempHodnik[i] = null;
      dodanaVozliscaLast++;
    }
    else if (tempHodnik[i][1] == dodanaVozlisca[st])
    {
      dodanaVozlisca[dodanaVozliscaLast] = tempHodnik[i][0];
      tempHodnik[i] = null;
      dodanaVozliscaLast++;
    }
  }
  st++;
}
//če je v tempHodnik še katero vozlišče, potem hodnik ni povezan
for (int i = 0; i < maxSten; i++)
  if (tempHodnik[i] != null)
```

```
return false;
```

Če je kombinacija sten dopustna, potem ta postane trenutno najboljša rešitev. Postopek nadaljujemo s preiskovanjem naslednje kombinacije in ga ponavljamo, dokler ne preverimo vseh kombinacij. Ko preverimo vse kombinacije z $maxSten$ stenami, se po istem postopku lotimo iskanja rešitve, kjer je v hodnik lahko vključenih največ $maxSten - 1$ sten. Če najdemo kakšno dopustno rešitev, ki je boljša od trenutno najugodnejše, potem to postane trenutno najboljša rešitev, iskanje rešitev pa bomo nadaljevali tudi za hodnike z $maxSten - 2$ sten. Postopek ponavljamo dokler dobivamo boljše rešitve. Algoritem zaključimo, ko pri nekem številu sten ne najdemo več dopustnih rešitev.

Sam potek algoritma iskanja najkrajšega hodnika je prikazan v nadaljevanju:

```
najDolzina = maxInt;
boolean nadaljuj = true;
while (nadaljuj)
{
  nadaljuj=false;
  while(not zadnjaKombinacija())
  {
    hodnik = NaslednjaKombinacijaSten();
    dolzina = vrniDolzino(hodnik);
    if(dolzina < najDolzina)
    {
      if (resitevDopustna(hodnik))
      {
        nadaljuj=true;
        najHodnik = hodnik;
        najDolzina = dolzina;
      }
```

```
    }
  }
}
```

5.3.3 Zaključek algoritma

Na koncu imamo rešitev problema podano kot množico sten, ki so vključene v hodnik. Vsaka stena je prestavljena kot enodimenzionalna tabela dolžine 3 , kjer je na prvem indeksu shranjeno prvo vozlišče, na drugem indeksu končno vozlišče, na tretjem indeksu pa je shranjena dolžina stene. Rešitev problema je torej množica sten, ki so vključene v hodnik. Vsota dolžin sten, ki so vključene v hodnik, predstavlja dolžino najkrajšega hodnika.

Poglavje 6

Aproksimacijski algoritmi

6.1 Algoritem 1 (najkrajše stene najprej)

Algoritem izhaja iz algoritma za izčrpno preiskovanje rešitev najkrajšega hodnika, opisanega v poglavju 5.3. V pripravi na algoritem smo omenili, da tabelo *stene* pred pričetkom algoritma sortiramo. Tako so najkrajše stene na začetku tabele. Sedaj pogledamo dolžino stene, ki je na indeksu *maxSten*-1 v tabeli *stene* in jo shranimo v spremenljivko *najDolzina*. Nato v tabelo *noveStene* prepišemo samo tiste stene, katerih dolžina je manjša ali enaka *najDolzina*. Algoritem sedaj pri iskanju rešitve ne bo več uporabljal tabele *stene*, ampak tabelo *noveStene*. Tako bomo pregledovali samo kombinacije, kjer je najdaljša stena dolga največ *najDolzina*. V hodnik je lahko vključenih največ *maxSten* sten.

Za vsako kombinacijo preverimo vsoto dolžin njenih sten. Če je dolžina večja ali enaka od trenutno najkrajše dolžine, potem nadaljujemo preiskovanje z naslednjo kombinacijo. Če je dolžina manjša, pa preverimo dopustnost rešitve. Preverjanje dopustnosti je enako kot pri izčrpnem preiskovanju, ki smo ga omenili v poglavju 5.3. Če je rešitev dopustna, jo shranimo v tabelo *najHodnik*.

Če ne najdemo rešitve, potem v tabeli *stene* poiščemo naslednjo dolžino stene, ki je večja od *najDolzina*. To vrednost zapišemo v spremenljivko *naj-*

Dolzina. Nato pobrišemo tabelo *noveStene* in vanjo zapišemo vse stene, katerih dolžina je manjša ali enaka *najDolzina*. Ponovno izvedemo algoritem za iskanje rešitve nad tabelo *noveStene*.

Postopek ponavljamo, dokler ob koncu iskanja rešitve nad tabelo *noveStene* ne najdemo dopustne rešitve. Na koncu je rešitev shranjena v tabeli *najHodnik* kot množica sten in njenih dolžin.

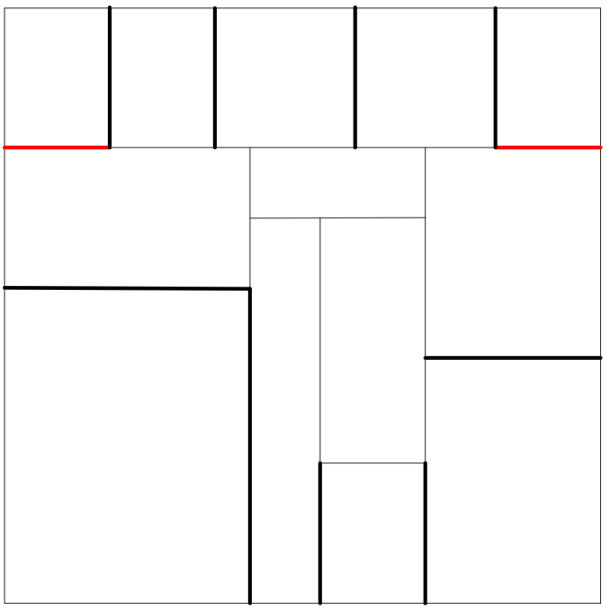
Algoritem je podoben algoritmu za izčrpno preiskovanje rešitev, le da ga izvajamo na neki podmnožici sten. Predvidevamo, da bo algoritem dober v primeru, ko je rešitev sestavljena iz več kratkih sten. Slabše se bo najbrž izkazal v primeru, če je v hodnik vključena tudi katera izmed daljših sten, saj bomo nekatere kombinacije večkrat preverjali. Rezultate testiranja algoritma bomo predstavili v poglavju 7.

6.2 Algoritem 2 (dodajanje najkrajših sten)

Ideja tega algoritma je, da sproti sestavljamo hodnik tako, da mu dodajamo najkrajšo steno, ki se dotika nastajajočega hodnika. Poleg tega mora v hodnik vključiti novo sobo. Če nobena stena k nastajajočemu hodniku ne prinese stika z novo sobo, potem dodamo najkrajšo izmed sten, ki še niso del hodnika. Za izhodišče vzamemo stene, ki imajo stik z zunanjim pravokotnikom.

6.2.1 Priprava za algoritem

Najprej izvedemo enako pripravo na algoritem, kot smo jo opisali v poglavju 5.3.1. Nato v tabelo *vstopneStene* shranimo stene, ki imajo natanko eno točko na zunanjem pravokotniku. Te stene bomo imenovali vstopne stene. Na sliki 6.1 so vstopne stene prikazane z odebeljenimi črnimi in rdečimi črtami. Izmed vseh vstopnih sten izberemo tiste, ki imajo najmanjšo dolžino in jih poimenujemo primerne vstopne stene, ki so na sliki 6.1 prikazane z odebeljeno rdečo črto. Primerne vstopne stene shranimo v tabelo *primerneVstopneStene*



Slika 6.1: Vstopne stene (odebeljene črne in rdeče črte) in primerne vstopne stene (odebeljene rdeče črte).

Potrebovali bomo tabelo, kamor bomo shranjevali stene, ki smo jih dodali v hodnik. Poimenovali jo bomo *hodnik*.

Vpeljemo tudi dve tabeli *dodanaVozlisca* in *dodaneSobe*. V *dodanaVozlisca* bomo vpisovali vozlišča, ki so že vključena v hodnik, v *dodaneSobe* pa sobe, katerih se hodnik že dotika. V nadaljevanju bomo ob dodajanju sten v *hodnik* dopolnjevali tudi ti dve tabeli in bomo imeli tu shranjena vozlišča oziroma sobe, ki jih hodnik že vsebuje.

Pripravimo tudi tabelo *najHodnik*, kjer bodo shranjene stene trenutno najkrajšega hodnika in spremenljivko *najDolzina*, kjer bo shranjena njegova dolžina. Tabela *najHodnik* je na začetku prazna, spremenljivka *najDolzina* pa dobi največjo možno vrednost.

6.2.2 Potek algoritma

Vzamemo prvo steno iz tabele *primerneVstopneStene* in jo dodamo v tabelo *hodnik*. V tabelo *dodanaVozlišča* vpišemo indeksa vozlišč, ki omejujeta prvo primerno vstopno steno. V tabelo *dodaneSobe* pa dodamo številke sob, ki se jih dotika.

Sedaj poiščemo najkrajšo možno steno, ki izpolnjuje naslednje tri pogoje:

1. stene še ni v tabeli *hodnik*,
2. z eno izmed sten, ki so že v tabeli *hodnik*, mora imeti vsaj eno skupno vozlišče (nova stena se mora dotikati hodnika),
3. vozlišče nove stene se dotika vsaj ene sobe, ki je še ni v tabeli *dodaneSobe*.

V primeru, da stena, ki ne ustreza vsem trem pogojem, ne obstaja, dodamo steno, ki ustreza samo prvemu in drugemu pogoju.

Ko dodamo steno v *hodnik*, posodobimo tudi tabeli *dodanaVozlisca* in *dodaneSobe*. V tabelo *dodanaVozlisca* dodamo številko novega vozlišča, v tabelo *dodaneSobe* pa dodamo številke sob, ki se jih dotika novo vozlišče (v kolikor sobe niso bile že predhodno dodane).

Postopek dodajanja sten ponavljamo, dokler se hodnik ne dotika vseh sob. Ko se hodnik dotika vseh sob, izračunamo dolžino poti. Če je dolžina manjša od *najDolzina*, potem hodnik prekopiramo v tabelo *najHodnik*, spremenljivka *najDolzina* pa dobi vrednost njegove dolžine.

Potem spremenljivke *dodanaVozlisca*, *dodaneSobe* in *hodnik* ponastavimo in postopek ponovimo še za ostale primerne vstopne stene.

Ko smo postopek ponovili za vse primerne vstopne stene, se algoritem zaključi. Kombinacija najkrajših sten je zapisana v tabeli *najHodnik*, njegova dolžina pa v spremenljivki *najDolzina*. Psevdokoda algoritma je predstavljena v nadaljevanju.

```
PripraviSpremenljivke();
```

```
//za vsako vstopno steno zgradimo hodnik
for (i = 0 to primerneVstopneStene.Length)
{
  for (int j=0 to stene.Length)
  {
    //ponastavimo spremenljivke za nov hodnik
    ponastaviSpremenljivke();
    while(true)
    {
       //pripravimo nadomestno steno,
       // če primerne stene ne bomo našli
       if (vozlisca.ImaStik(stene[i]))
       {
         if (stene[i][2] < najDolzinaNadomestna)
         {
           najStenaNadomestna = j;
           najDolzinaNadomestna = stene[i][2];
         }
         //poiščemo primerno steno
         if (stene[i][2] < najDolzina)
         {
           //preverimo, če z dodajanjem stene pridobimo stik
           // z novo sobo
           boolean dodaj = PridobimoNovoSobo(stene[i]);
           if(dodaj)
           {
             najDolzina = stene[i][2];
             najStena = i;
           }
         }
       }
```

```
    }
    if (obstaja(najStena)) //če smo našli ustrezno steno
    {
       hodnik.add(stene[najStena]);
       //dodamo na novo pridobljeno vozlišče
       dodanaVozlisca.add(stene[najStena]);
       //dodamo na novo pridobljene sobe
       dodaneSobe.add(DodajNoveSobe(stene[najStena]);
    }
    else
    {
       hodnik.add(stene[najStenaNadomestna]);
       //dodamo na novo pridobljeno vozlišče
       dodanaVozlisca.add(stene[najStenaNadomestna]);
    }
    //če je rešitev dopustna, zaključimo iskanje
    if (HodnikDopusten(hodnik)) break;
}
```

6.3 Algoritem 3 (ocenjevanje primernosti vozlišča)

Algoritem, opisan v prejšnjem poglavju, je v vsakem koraku k hodniku dodajal najboljše stene, dokler nismo obiskali vseh sob. Namen tega algoritma pa je, da za vsako vozlišče izračunamo neko oceno vozlišča in nato najboljše vozlišče dodamo v nastajajoči hodnik. Tu torej nimamo opravka s stenami, ampak z vozlišči. Prav tako ni potrebno, da se vozlišča ob gradnji stikajo med seboj.

6.3.1 Ideja algoritma

Za vsako izmed vozlišč, ki ležijo na zunanjem pravokotniku, bomo sestavili hodnik. Nato bomo izmed vseh hodnikov izbrali tistega, ki bo imel najmanjšo dolžino. Na začetku iskanja najkrajšega hodnika za vsako vozlišče izračunamo njegovo oceno. Nato vozlišče z najvišjo oceno dodamo v hodnik in posodobimo ocene. Vozlišča bomo dodajali tako dolgo, dokler ne pridemo do dopustne rešitve. Potem pogledamo, če lahko katero izmed vozlišč odstranimo tako, da je hodnik še vedno dopusten. Če taka vozlišča obstajajo, jih odstranimo, saj s tem skrajšamo dolžino hodnika. Po koncu krajšanja hodnika dobimo rešitev za neko vozlišče, ki leži na zunanjem pravokotniku. Nato poiščemo rešitve še za ostala vozlišča iz zunanjega pravokotnika in izmed teh rešitev nato poiščemo najkrajšo, ki je tudi končna rešitev algoritma.

Ocena vsakega vozlišča je sestavljena iz treh enakovrednih delov:

- vsota dolžin vseh sten, ki se stikajo v vozlišču,
- oddaljenost vozlišča od najbližjega vozlišča, ki je že del hodnika,
- število novih sob, ki jih pridobimo z dodajanjem vozlišča v hodnik.

Vsoto dolžin sten dobimo tako, da seštejemo dolžine vseh sten, ki se stikajo v vsakem izmed vozlišč. Nato poiščemo najkrajšo in najdaljšo steno v grafu. Najdaljša stena je vredna 0 enot, najkrajša pa 100. Ocena sten, katerih dolžina je večja od najkrajše in manjša od najdaljše, je sorazmerna glede na njihovo vsoto dolžin.

Oddaljenost vozlišča od najbližjega vozlišča, ki je že del hodnika, izračunamo tako, da za vsako vozlišče izračunamo, koliko vozlišč je oddaljen od najbližjega vozlišča, ki je že del hodnika. Oceno nato določimo s pomočjo tabele 6.1.

V tabelo *dodaneSobe* shranjujemo številke sob, ki se jih vozlišča v hodniku že dotikajo. Za vsako vozlišče preverimo, koliko sob, ki jih še ni v tabeli *dodaneSobe*, se dotika. Če se vozlišče dotika več novih sob, ima višjo oceno. Vozlišča, ki k hodniku ne prinesejo novih sob, so vredna najmanj. Oceno določimo s pomočjo tabele 6.2.

število vozlišč na poti	ocena
0	100
1	50
2	25
3 ali več	0

Tabela 6.1: Ocena vozlišča glede na oddaljenost

število novih sob	ocena
4	100
3	66
2	50
1	33
0	0

Tabela 6.2: Ocena vozlišča glede na število novih sob

6.3.2 Priprava na algoritem

Na začetku napolnimo tabeli *stene* in *sobe* ter nastavimo vse začetne spremenljivke tako, kot je opisano v poglavju 5.3.1.

Kot smo že omenili v prejšnjem poglavju, je ocena vsakega vozlišča sestavljena iz treh enakovrednih delov. Ocena vsot dolžine vseh sten, ki se stikajo v vozlišču, je ves čas konstantna, ostali dve oceni pa se med gradnjo hodnika spreminjata. Zato lahko oceno vsot dolžine vseh sten izračunamo na začetku in nam potem tega dela ocene ni treba spreminjati, saj se dolžine sten ne spreminjajo.

Ocene bomo shranjevali v dvodimenzionalno tabelo *ocene* velikosti *stVozlisc* x 4. Za vsako vozlišče bo na indeksu 0 shranjena vsota povezav, na indeksu 1 ocena oddaljenosti, na indeksu 2 ocena števila novih sten. Na indeksu 3 pa bomo hranili skupno oceno vozlišča, ki jo izračunamo tako, da vsako izmed prejšnjih treh ocen pomnožimo z 1/3 in nato zmnožke seštejemo. Vse ocene zaokrožimo na cela števila. Skupna ocena je tako kot vse ostale ocene celo število med 0 in 100.

6.3.3 Potek algoritma

Za vsako izmed vstopnih vozlišč bomo zgradili najkrajši hodnik in nato izmed vseh izbrali najkrajšega.

Na začetku dodamo v *hodnik* začetno vozlišče iz tabele *vstopi* in v tabelo *dodaneSobe* zapišemo sobi, ki se stikata z začetnim vozliščem. Nato za vsako vozlišče izračunamo oddaljenost in število novih sob ter vrednosti pretvorimo v ocene tako, kot je to opisano v poglavju 6.3.1. Nato poiščemo vozlišče z najvišjo oceno in ga dodamo v *hodnik*. Sedaj se lotimo preverjanja dopustnosti rešitve. Ta postopek je pri tem algoritmu drugačen, saj imamo tu opravka z vozlišči in ne s stenami.

Preverjanje dopustnosti rešitve pričnemo s preverjanjem, če hodnik vsebuje vsaj eno izmed vstopnih vozlišč, ki so shranjena v tabeli *vstopi*. Če ga vsebuje, potem preverimo, če ima vsaka soba stik z vsaj enim izmed vozlišč v hodniku. Če ga ima, pride na vrsto najtežji del preverjanja. Ugotoviti moramo namreč, če je možno vsa vozlišča povezati med seboj. Najprej hodnik shranimo v začasno tabelo *tempHodnik*. Nato vzamemo vstopno točko iz *tempHodnik* in jo dodamo v novo tabelo *obiskanaVozlisca* in jo zatem izbrišemo iz *tempHodnik*. Sedaj izmed preostalih vozlišč skušamo najti steno, ki ima eno vozlišče v tabeli *obiskanaVozlisca*, drugo pa v *tempHodnik*. Če jo najdemo, jo dodamo v *obiskanaVozlisca* in izbrišemo iz *tempHodnik*. Postopek ponavljamo, dokler ni mogoče dodati nobene stene. Sedaj preverimo začasno tabelo *tempHodnik*. Če je tabela prazna, to pomeni, da smo lahko med seboj povezali vsa vozlišča in je rešitev torej dopustna. Če tabela ni prazna, potem so nekateri deli hodnika nepovezani med seboj in rešitev ni dopustna.

Iskanje rešitve se končna, ko pridemo do prve dopustne rešitve. Nato se lotimo še optimizacije rešitve, saj lahko hodnik vsebuje odvečna vozlišča. Najprej v *tempHodnik* shranimo trenutni hodnik, ki je shranjen v tabeli *hodnik*. Nato iz *hodnika* odstranimo prvo vozlišče in preverimo dopustnost rešitve. Če je rešitev dopustna, potem shranimo skrajšan hodnik v novo spremenljivko *tempHodnik* in njegovo dolžino v *tempDolzina*. Nato vzamemo zopet prvotni

hodnik in iz njega odstranimo drugo vozlišče. Preverimo dopustnost rešitve. Če je rešitev dopustna in je dolžina novega hodnik manjša od *tempDolzina*, potem v *tempHodnik* shranimo nov hodnik in v *tempDolzina* njegovo dolžino. Postopek ponavljamo, dokler ne preverimo vseh vozlišč. Na koncu imamo v *tempHodnik* hodnik z najmanjšo dolžino po odstranjenem enem vozlišču. Sedaj postopek ponovimo za hodnik *tempHodnik* in ga ponavljamo, dokler po odstranjevanju vozlišča še vedno dobimo dopustno rešitev. Odstranjevanje vozlišča izvajamo s pomočjo rekurzije. Na koncu postopka imamo najboljši hodnik shranjen v tabeli *tempHodnik*.

Sedaj celoten postopek ponovimo še za vsa ostala vstopna vozlišča in tako na koncu dobimo končno rešitev.

Psevdokoda algoritma je predstavljena v nadaljevanju.

```
IzracunajVsotoDolzinSten(ocene);
najDolzina=int.maxInt;
for (i = 0 to vstopi.Length)
{
  //ponastavimo spremenljivke za nov hodnik
  ponastaviSpremenljivke();
  DodajPrvoVozlisce();
  while(true)
  {
    IzracunajOddaljenosti(ocene);
    IzracunajSteviloNovihSob(ocene);
    IzracunajSkupnoOceno(ocene);
    najVozlisce = NajboljeVozlisce(ocene);
    hodnik.Add(najVozlisce);
    dodaneSobe = ObiskaneSobe(hodnik);
    if(dodaneSobe.count < stSob)
      continue;
    if (HodnikDopusten(hodnik))
```

```
      break;
  }
  //odstranimo odvečna vozlišča
  hodnik = OdstraniOdvecnaVozlisca(hodnik);
  tempDolzina = vrniDolzino(hodnik);
  if (tempDolzina < najDolzina)
  {
    najHodnik = hodnik;
    najDolzina = tempDolzina;
  }
}
```

Poglavje 7

Testiranje in rezultati delovanja algoritma

Testiranje smo izvedli na osebnem računalniku z naslednjimi lastnostmi:

frekvenca procesorja:	2,40 GHz
število jeder na procesorju:	4
količina pomnilnika:	4 GB
tip pomnilnika:	DDR2
frekvenca pomnilnika:	800 MHz
operacijski sistem:	Windows 7 Professional 64-bit

Testirali smo delovanje štirih algoritmov, ki so bili predstavljeni v tej diplomski nalogi. Eden nam vedno vrne optimalno rešitev, trije pa so aproksimacijski in nam ne vrnejo vedno optimalne rešitve.

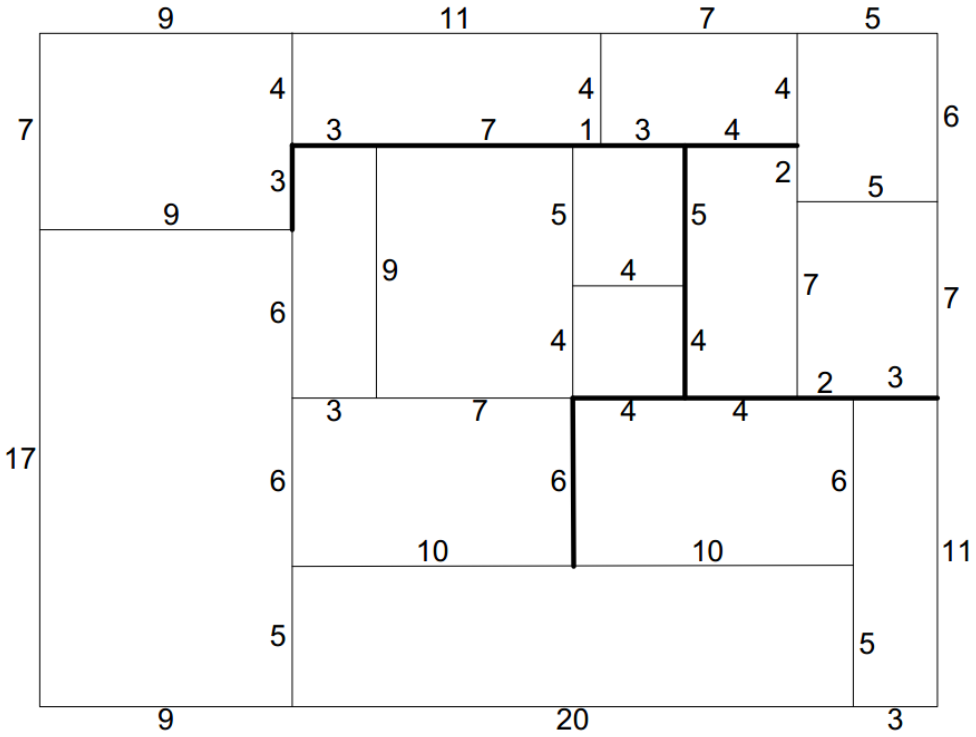
- **Algoritem za minimizacijo dolžine hodnika - oznaka NOSN**
- **Algoritem 1 (najkrajše stene najprej) - oznaka ASTE**
- **Algoritem 2 (dodajanje najkrajših sten) - oznaka ADOD**
- **Algoritem 3 (ocenjevanje primernosti vozlišča) - oznaka AOCE**

7.1 Rezultati testiranja

Algoritme smo testirali na štirih primerih, ki so prikazani na slikah 7.1, 7.2, 7.3 in 7.4. Številke ob stenah predstavljajo njihovo dolžino. Na vsakem primeru je najkrajši hodnik predstavljen z odebeljeno črno črto.

Primer 1:

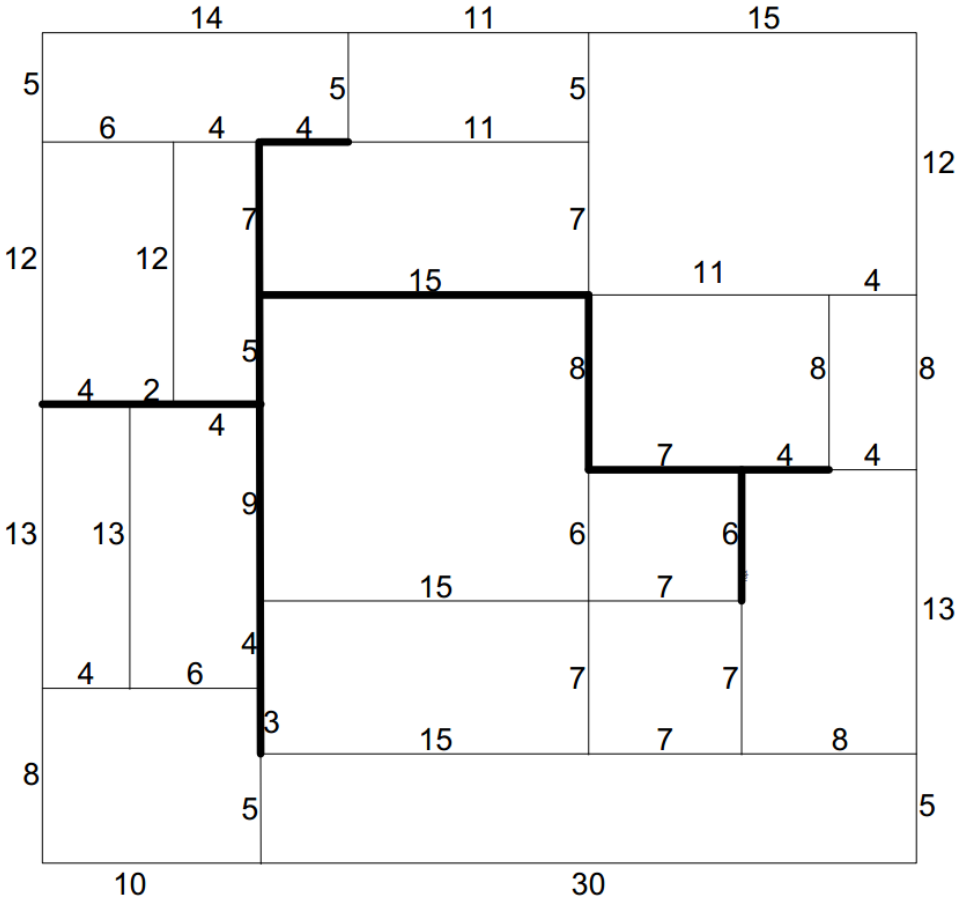
- Število sten: 41
- Število sob: 15
- Število vozlišč: 27



Slika 7.1: Testni primer 1

Primer 2:

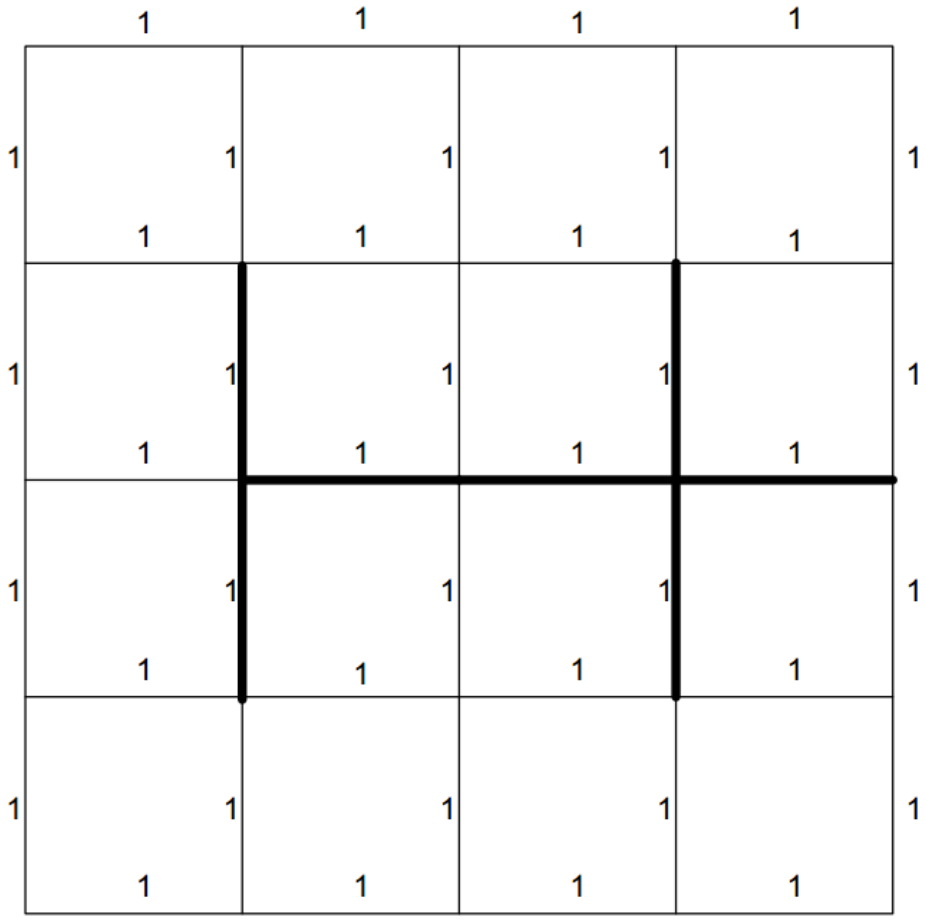
- Število sten: 46
- Število sob: 17
- Število vozlišč: 30



Slika 7.2: Testni primer 2

Primer 3:

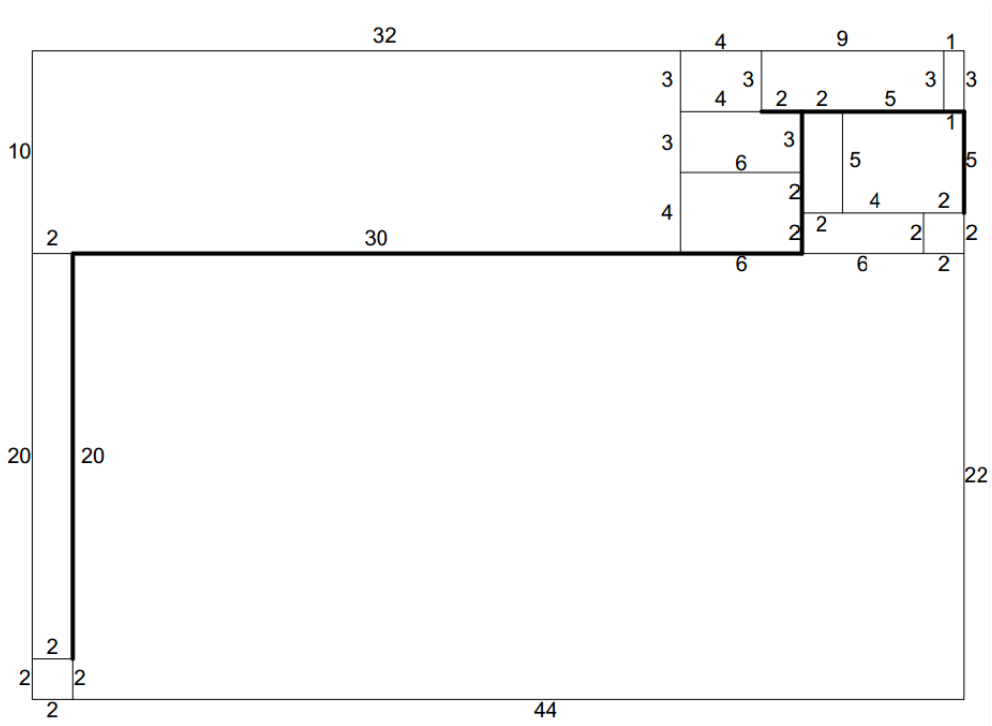
- Število sten: 36
- Število sob: 16
- Število vozlišč: 21



Slika 7.3: Testni primer 3

Primer 4:

- Število sten: 36
- Število sob: 13
- Število vozlišč: 24



Slika 7.4: Testni primer 4

Primer 1 in primer 2 predstavljata naključno razporeditev sob. Primer 3 prikazuje sobe v obliki enako velikih kvadratov, ki so razporejeni v obliki šahovnice. Primer 4 pa predstavlja problem, kjer imamo več majhnih sob, ki se držijo skupaj in eno majhno sobo, do katere pridemo preko daljših sten. Menimo, da bi nekateri aproksimacijski algoritmi lahko na tem primeru vrnili slabše rezultate.

V tabelah 7.1, 7.2, 7.3 in 7.4 so prikazani rezultati testiranja vseh štirih algoritmov na vseh štirih primerih.

	NOSN	ASTE	ADOD	AOCE
čas izvajanja [s]	10557	798	0,011	0,144
dolžina hodnika	49	49	51	50

Tabela 7.1: Rezultati testiranja algoritmov na primeru 1

	NOSN	ASTE	ADOD	AOCE
čas izvajanja [s]	260943	179957	0,014	0,146
dolžina hodnika	82	89	85	90

Tabela 7.2: Rezultati testiranja algoritmov na primeru 2

	NOSN	ASTE	ADOD	AOCE
čas izvajanja [s]	235	72	0,016	0,065
dolžina hodnika	7	9	9	7

Tabela 7.3: Rezultati testiranja algoritmov na primeru 3

	NOSN	ASTE	ADOD	AOCE
čas izvajanja [s]	5354	6757	0,010	0,232
dolžina hodnika	78	80	98	78

Tabela 7.4: Rezultati testiranja algoritmov na primeru 4

Algoritem 2 in algoritem 3 smo testirali vsakega po desetkrat na vsakem primeru in smo kot rezultat napisali povprečje vseh desetih meritev. Zaradi kratkega časa izvajanja je prišlo do manjših odstopanj pri posameznih meritvah.

7.2 Komentar rezultatov

Izračun najkrajše dolžine hodnika je računsko zahtevna operacija. Pri primeru 1 vidimo, da je izračun trajal 10557 sekund, kar je nekaj manj kot tri ure. V primeru 2 smo testirali delovanje algoritmov na nekoliko večjem problemu. Problem je bil večji za 5 sten, 2 sobi in 3 vozlišča. Izračun je

pri primeru 2 trajal 260943 sekund, kar znaša več kot 72 ur. Ugotavljamo, da čas iskanja najkrajše dolžine hodnika zelo hitro narašča in hitro postane neobvladljiv. Kako pa so se izkazali aproksimacijski algoritmi?

7.2.1 Algoritem 1

Opazimo, da se je algoritem 1 glede časa izvajanja odrezal najslabše izmed vseh aproksimacijskih algoritmov. Pri primerih 1, 2 in 3 je izračun rešitve časovno krajši od izračuna optimalnega hodnika, pri primeru 4 pa izračun traja celo dlje. Dolžina hodnika je v primeru 1 enaka optimalni, v ostalih primerih pa vedno slabša.

7.2.2 Algoritem 2

Algoritem 2 je izmed vseh algoritmov na testnih primerih potreboval najmanj časa za izračun hodnika. Izračunana dolžina hodnika je v primerih 1,3 in 4 najdaljša. Opazimo, da ima algoritem velike težave pri primeru 4, saj izračunana dolžina hodnika močno odstopa v negativno smer.

7.2.3 Algoritem 3

Algoritem 3 je po časovni zahtevnosti na drugem mestu pri vseh testnih primerih. Glede izračunane dolžine hodnika je samo v primeru 2 na zadnjem mestu. V dveh primerih se je izkazalo, da je izračunana dolžina hodnika enaka optimalni.

7.2.4 Povzetek testiranja

Izmed vseh treh aproksimacijskih algoritmov se je časovno najslabše odrezal algoritem 1. Brez oklevanja ga lahko zavržemo, saj je v primeru 4 potreboval celo več časa za izračun kot optimalni algoritem. Algoritem 2 in algoritem 3 sta oba časovno učinkovita, saj nam hitro vrneta rešitev. Pri tem lahko damo manjšo prednost algoritmu 3, saj je v treh od štirih primerov izračunal krajši

hodnik. Njegova prednost je, da pred zaključkom poteka skuša odstraniti odvečna vozlišča, kar v nekaterih primerih prinese boljši rezultat.

Poglavje 8

Sklepne ugotovitve

V tej diplomski nalogi smo predstavili problem minimizacije dolžine hodnika. Razvili smo algoritem za izčrpno preiskovanje rešitev najkrajšega hodnika. Ker je problem časovno zahteven, smo razvili tudi tri aproksimacijske algoritme. Dva izmed njih sta se pri testiranju dobro izkazala in nam v kratkem času vrnila dobre rezultate.

Če bi v omrežje morali povezati 10-15 računalnikov, potem se nam splača uporabiti algoritem za izčrpno preiskovanje rešitev, saj nam na rezultat ne bo treba predolgo čakati. Če pa gradimo večje omrežje z več računalniki, gradimo večjo stavbo z več prostori ali napeljujemo optično omrežje v mestu z več stavbami, potem je smiselno uporabiti katerega izmed aproksimacijskih algoritmov, saj bi za izračun optimalne rešitve potrebovali preveč časa.

Vsakega izmed algoritmov je mogoče še izboljšati s pomočjo človeka. V nekaterih primerih namreč človek ob pogledu na tloris problema hitro vidi, katere stene lahko z veliko verjetnostjo izključimo iz problema. Računalniški program tega znanja nima. S tem se nam močno skrajša čas preiskovanja.

Prav tako bi bilo mogoče algoritme prirediti za določeno vrsto problema kot na primer za primer, kjer so vse sobe približno enake velikosti, kjer so nekatere majhne sobe oddaljene od ostalih, kjer imamo veliko vstopnih točk, ... Poleg navedenega pa bi se v okviru nadaljnjega dela bilo potrebno posvetiti razvoju aproksimacijskih algoritmov za ostale različice problema, omenjene

v poglavju 2.2.3.

Slike

Tabele

Literatura

[1] S. Arora, "Approximation schemes for NP-hard geometric optimization problems: A survey. Mathematical Programming" v Mathematical Programming, str. 97:43-69,

[2] S. Arora, "Nearly Linear Time Approximation Schemes for Euclidean TSP and other Geometric Problems", Journal od the ACM, str. 45:1-30, 1998

[3] H. L. Bodlaender, C. Feremans, A. Grigoriev, E. Penninkx, R. Sitters, T. Wolle,"On the minimum corridor connection problem and other generalized geometric problems" v Approximation and Online Algorithms, T. Erlebach in C. Kaklamanis, Zurich, 2006, str. 69-82.

[4] E. D. Demaine, J. O'Rourke, "Open problems from CCCG 2000" v Proceesings of the 13th Canadian Conference on Computional Geometry (CCCG 2001), Ontario, 2001, str. 185-187

[5] D. Eppstein, "Some open problems in graph theory and computional geometry", November, 2001.

[6] A. Gonzalez-Gutierrez, "Minimum-Length Corridors: Complexity and Approximations", Santa Barbara, September, 2007.

[7] A. Gonzalez-Gutierrez, T. F. Gonzalez, "Complexity of the minimum-length corridor problem", Santa Barbara, 2006

[8] P. Slavik, “Approximation Algorithms for Set Cover and Related Problems”, Buffalo, 1998

[9] P. Slavik, “The Errand scheduling problem”, Buffalo, 1997.

[10] (2012) Introduction to the C# Language and the .NET Framework. Dostopno na: http://msdn.microsoft.com/library/z1zx9t92

[11] (2012) .NET Framework. Dostopno na: http://msdn.microsoft.com/en-us/vstudio/aa496123

[12] (2012) Visual Studio Professional 2008. Dostopno na: http://msdn.microsoft.com/en-us/vstudio/aa700830.aspx

[13] (2012) Approximate Algorithms. Dostopno na: http://www.personal.kent.edu/ rmuhamma/

[14] (2012)The Traveling salesman Problem. Dostopno na: http://www.tsp.gatech.edu/index.html

[15] (2012)Travelling salesman problem. Dostopno na: http://en.wikipedia.org/wiki/Travelling_salesman_problem

[16] (2005) Group Steiner Tree. Dostopno na: http://www.cs.cmu.edu/