UNIVERZITA KOMENSKÉHO V BRATISLAVE

FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY

Robot Tatrabot rieši hru Sokoban

Bakalárska práca

2017 **Alex Mendel**

UNIVERZITA KOMENSKÉHO V BRATISLAVE
FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY

Robot Tatrabot rieši hru Sokoban

Bakalárska práca

Študijný program:	Aplikovaná informatika
Študijný odbor:	2511 Aplikovaná informatika
Školiace pracovisko:	Katedra aplikovanej informatiky
Školiteľ:	Mgr. Pavel Petrovič, PhD.

Bratislava, 2017 **Alex Mendel**

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Alex Mendel
Študijný program: aplikovaná informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor: 9.2.9. aplikovaná informatika
Typ záverečnej práce: bakalárska
Jazyk záverečnej práce: slovenský
Sekundárny jazyk: anglický

Názov: Robot Tatrabot rieši hru Sokoban
Tatrabot Robot Solves Sokoban Game

Cieľ: Tatrabot je riadený 32-bitovým jednočipovým počítačom rady STM32, má bohatú senzorickú výbavu: 3 osí akcelerometer, gyroskop, magnetometer, barometer, teplomer, 2 IR senzory na detekciu prekážok, SHARP IR senzor na meranie vzdialenosti, 4-kanálový senzor na čiaru, zvukový výstup, tlačidlá, signálne LED, otáčkové senzory, BlueTooth. V tejto práci sa zameriame na riešenie úloh presúvania kociek v labyrinte - úloha podobná hre Sokoban a simultánnu vizualizáciu jeho činnosti. Študent navrhne a implementuje algoritmus na riešenie hry a pokúsi sa ho prepojiť s reálnym mobilným robotickým systémom.

Literatúra: Vladimír Šatura: SBOT Sokoban, bakalárska práca, FMFI UK, 2010.

Kľúčové slová: mobilný robot, sokoban, prehľadávanie stavového priestoru

Vedúci: Mgr. Pavel Petrovič, PhD.
Katedra: FMFI.KAI - Katedra aplikovanej informatiky
Vedúci katedry: prof. Ing. Igor Farkaš, Dr.
Dátum zadania: 01.10.2015

Dátum schválenia: 28.10.2015

doc. RNDr. Damas Gruska, PhD.
garant študijného programu

....
študent

....
vedúci práce

ČESTNÉ VYHLÁSENIE

Čestne vyhlasujem, že som bakalársku prácu „Robot Tatrabot rieši hru Sokoban" vypracoval samostatne s použitím uvedenej literatúry a zdrojov dostupných na internete.

V Bratislave dňa 25.5.2017

……………………………………..........

Alex Mendel

Poďakovanie

Touto cestou by som rád poďakoval svojmu školiteľovi Mgr. Pavlovi Petrovičovi, PhD. za užitočné rady, pripomienky, trpezlivosť a usmernenie pri písaní mojej bakalárskej práce.

Bratislava 25.5.2017 Alex Mendel

ABSTRAKT

Cieľom tejto práce je navrhnúť a implementovať algoritmus na riešenie hry a jeho prepojenie s reálnym mobilným robotickým systémom. Tatrabot je riadený 32-bitovým jednočipovým počítačom rady STM32, má bohatú senzorickú výbavu: 3 osí akcelerometer, gyroskop, magnetometer, barometer, teplomer, 2 IR senzory na detekciu prekážok, SHARP IR senzor na meranie vzdialenosti, 4-kanálový senzor na čiaru, zvukový výstup, tlačidlá, signálne LED, otáčkové senzory, BlueTooth. V tejto práci sa zameriame na riešenie úloh presúvania kociek v labyrinte - úloha podobná hre Sokoban a simultánnu vizualizáciu jeho činnosti.

kľúčové slová: mobilný robot, sokoban, prehľadávanie stavového priestoru

ABSTRACT

The aim of this work is to design and implement algorithm for game solving and link it to real mobile robotic system. Tatrabot is controlled by 32-bit single-chip computer from STM32 series. It has huge sensory equipment: 3-axis accelerometer, gyroscope sensor, magnetometer, barometer sensor, thermometer, 2 IR sensors for obstacle detection, SHARP IR sensor for distance measurement, 4-channel sensor for line detection, audio output, buttons, LEDs for signalisation, speed sensors, BlueTooth. In this work we will aim for solving tasks of cubes movement in labyrinth – task is similar to Sokoban game and simultaneous visualisation of its activity.

key words: mobile robot, sokoban, searching in state space

OBSAH

Úvod...1

1 Východiská 2

1.1 Prehľadávanie stavového priestoru 2

1.1.1 Prehľadávanie do hĺbky (Depth-first search)......................................4

1.1.2 Prehľadávanie do šírky (Breadth -first search)....................................4

1.1.3 Prehľadávanie hladným algoritmom (Greedy algorithm).......................5

1.1.4 Prehľadávanie A* (A star algorithm) 5

1.2 Tatrabot 6

1.3 Sokoban 9

1.4 Sokoban solver...11

2 Návrh riešenia 12

2.1 Prvá verzia programu v C++...14

2.2 Druhá verzia programu v C 16

2.3 Návrh pohybu robota po mape...19

3 Implementácia..20

3.1 Prvá verzia implementácie algoritmu prehľadávania v C++ 20

3.2 Druhá verzia implementácie algoritmu prehľadávania v C 25

3.3 Implementácia pohybu robota po labyrinte 27

Záver 31

Zdroje a použitá literatúra 32

Úvod

V posledných rokoch môžeme sledovať rýchly rozvoj moderných technológii, umelej inteligencie a sledujeme tendenciu modernizácie a automatizácie systémov a spotrebičov v našich domácnostiach, školách a pracoviskách. Stále stúpa nahradzovanie pracovníkov automatizovanými strojmi a robotmi, ktorí dokážu riešiť rozličné situácie a spolupracovať vo výrobných procesoch. Táto práca sa venuje práve implementácii algoritmov pomocou umelej inteligencie robota.

Cieľom mojej bakalárskej práce je navrhnúť a následne implementovať riešenie pre hru Sokoban pomocou robota Tatrabot. Zameriame sa na riešenie úloh presúvania kociek v labyrinte (Sokoban) a simultánnu vizualizáciu jeho činnosti.

Táto práca je rozčlenená na niekoľko kapitol.

Prvá, východisková kapitola obsahuje prehľad použitých technológií, prehľad algoritmov a vysvetlenie teórie, ktoré môžu pomôcť pri riešení problému prehľadávania grafov a implementácii pomocou robota Tatrabot.

V druhej kapitole sa budeme venovať návrhom dátových štruktúr a zároveň aj návrhom algoritmov, ktoré sú potrebné na vyriešenie problému prehľadávania stavového priestoru a aj algoritmom, ktoré využije Tatrabot na to aby sa dokázal pohybovať po labyrinte.

V poslednej implementačnej kapitole zhrnieme postup implementácie navrhnutých dátových štruktúr aj samotných algoritmov. Popíšeme aj niektoré zaujímavé problémy, ktoré sa počas implementácie vyskytli.

1 Východiská

V tejto kapitole sa venujem prehľadu potrebnej teórie a algoritmom, ktoré neskôr použijem pri implementácii a tiež opisujem výhody a nevýhody existujúcich algoritmov. Ďalej sa venujem samotnému robotovi Tatrabotovi a na záver kapitoly opíšem hru Sokoban.

1.1 Prehľadávanie stavového priestoru

Nasledovná kapitola popisuje metódy prehľadávania stavového priestoru podľa zdrojov [1], [4], [5] a [6]

Stavový priestor je vlastne množina rozličných stavov, ktoré môžu nastať pri riešení problému prehľadávania grafov, z ktorých jeden stav je začiatočný a niektoré sú cieľové. Sledovaním určitých udalostí sa dá v stavovom priestore prejsť z jedného stavu do iného stavu. Môžeme si to predstaviť ako orientovaný graf, ktorého vrcholy sú stavy a hrany medzi vrcholmi sú udalosti, ktoré menia predošlý stav na nejaký nový stav. Riešenie problému potom získame nájdením cesty zo začiatočného vrcholu do cieľového vrcholu.

Tento stavový priestor však môže byť veľmi rozsiahly a nájdenie cesty z počiatočného stavu do cieľového stavu môže byť v reálnom čase takmer nemožné, pretože jedno-čipový počítač obsiahnutý v Tatrabotovi má obmedzenú operačnú pamäť. Z tohto dôvodu je použitie jednoduchých metód prehľadávania nevyhovujúce.

Existuje mnoho procedúr a algoritmov, pomocou ktorých môžeme prehľadávať stavový priestor avšak kvôli vyššie popísaným dôvodom (obmedzená veľkosť RAM) má každá svoje výhody a nevýhody.

Procedúry prehľadávania môžeme rozdeliť na informované a neinformované (slepé). Neinformované procedúry prehľadávania nemajú k dispozícii žiadne vhodné znalosti o stavovom priestore, ktoré by im pomohli urýchliť nájsť cestu k cieľovému stavu. Musia preto systematicky prechádzať všetky vrcholy, kým nenájdu riešenie. Jednotlivé algoritmy sa líšia len systémom akým vykonávajú toto prehľadávanie.

Neinformované procedúry :
Prehľadávanie do šírky
Prehľadávanie do hĺbky
Prehľadávanie do hĺbky s obmedzením
Obojsmerné prehľadávanie

Na rozdiel od neinformovaných procedúr informované metódy majú informáciu o stavovom priestore a vďaka nej vedia odhadovať, ktoré stavy prehľadať ako prvé, čo im umožňuje urýchliť čas potrebný na nájdenie riešenia a zároveň aj znížiť potrebnú pamäť.
Tento odhad reprezentuje takzvaná heuristická funkcia $h(n)$, ktorá dokáže určiť stavy, ktoré sú bližšie k riešeniu a je väčšia šanca, že práve cez tieto stavy je potrebné hľadať cestu k cieľu.
Čím nižšie hodnoty funkcia $h(n)$ nadobúda, tým je pravdepodobnejšie, že cesta povedie práve cez tento stav n. Problémom však je, že táto funkcia musí byť správna, pretože čím lepšia heuristika je k dispozícii, tým rýchlejšie a s menším zaťažením pamäte nájde správne riešenie.

Informované procedúry:
Greedy algoritmus
Horolezecký algoritmus
Algoritmus A*
Prehľadávanie lúčom

Výstup vyššie uvedených metód a algoritmov porovnávame pomocou štyroch kritérií:

- **Kompletnosť riešenia –** určuje či spomenuté algoritmy a procedúry prehľadávania dokážu zabezpečiť nájdenie správneho riešenia, ak nejaké existuje.

- **Optimálnosť riešenia –** kvalita nájdeného riešenia je určená pomocou ceny cesty. Optimálne riešenie najlacnejšiu cestu (najmenší počet krokov) spomedzi všetkých ostatných riešení.

- **Pamäťová zložitosť** – určuje koľko operačnej pamäte daná procedúra alebo algoritmus potrebuje aby vykonal prehľadávanie a vyriešil problém. Môžeme ju určiť ako počet kilobytov (kB), alebo ako minimálny alebo maximálny počet stavov, ktoré potrebujeme mať uchované v pamäti.

- **Časová zložitosť** – určuje koľko času potrebujeme na to, aby sme našli riešenie. Môže sa jednať o minimálny, maximálny alebo priemerný čas. Čas je však veličina, ktorá úzko súvisí s výkonom počítača na ktorom výpočet prebieha. Preto sa dá určiť aj ako počet stavov, ktoré je potrebné prejsť aby sme dosiahli riešenie.

1.1.1 Prehľadávanie do hĺbky (Depth-first search)

Pri prehľadávaní do hĺbky ide o úplnú metódu prehľadávania grafu. Avšak táto metóda nemusí byť optimálna. Algoritmus vždy prehľadáva prvého nasledovníka vrcholu, v ktorom začal, až kým sa nedostane na najhlbšiu úroveň stromu. Táto metóda začne rekurzívne prehľadávať vyššie úrovne stromu len v prípade, ak príde do úrovne, v ktorej už vrchol nemá nasledovníka a nie je to cieľový vrchol. Na začiatku prehľadávania sú všetky vrcholy označené ako neprehľadané, až kým sa algoritmus backtrackingom cez ne nevráti, vtedy sa vrcholy označia ako VISITED a viac sa cez ne neprehľadáva.

Prehľadávanie do hĺbky má z hľadiska pamäťovej zložitosti veľmi nenáročné požiadavky. Z hľadiska časovej zložitosti je to exponenciálna zložitosť v závislosti od hĺbky hľadania $O(b^d)$. Avšak, v praxi je prehľadávanie do hĺbky často rýchlejšie ako do šírky a to hlavne pri problémoch, ktoré majú veľa riešení.

1.1.2 Prehľadávanie do šírky (Breadth -first search)

Pri prehľadávaní do šírky sa jedná o jednoduchú metódu prehľadávania stavového priestoru. Algoritmus postupne prehľadáva všetky vrcholy tak, že najskôr pozrie všetkých nasledovníkov prvého vrcholu, potom všetkých nasledovníkov týchto vrcholov a tak ďalej.

Vo všeobecnosti teda môžeme povedať, že všetky vrcholy v hĺbke d sú prehľadané pred vrcholmi v hĺbke d+1. Prehľadávanie do šírky je veľmi systematická metóda, pretože najskôr uvažuje cesty dĺžky 1, potom cesty dĺžky 2 a tak ďalej.

Podľa vyššie uvedených štyroch kritérií je prehľadávanie do šírky kompletné a optimálne. Z hľadiska výpočtovej zložitosti je pamäťová aj časová zložitosť exponenciálna v závislosti od hĺbky hľadania ***O(b^d)***. V praxi to znamená, že pre veľké problémy nám nebude stačiť pamäť.

1.1.3 Prehľadávanie hladným algoritmom (Greedy algorithm)

Greedy algoritmus je jeden z najjednoduchších informovaných metód prehľadávania. Cieľom tohto algoritmu je minimalizovať zostávajúcu cenu cesty k dosiahnutiu cieľového vrcholu. Teda vrchol, ktorého stav je zvážený ako najbližší k cieľovému stavu sa vždy navštívi prvý. Vo väčšine prípadov je cena dosiahnutia cieľu iba približne určená. Funkcia, ktorá počíta hodnotu zostávajúcich krokov do cieľa sa nazýva heuristická funkcia (označujeme *h*).

Najznámejšie greedy algoritmy sú Primov algoritmus a Kruskalov algoritmus na hľadanie minimálnej kostry v grafe. Pri Primovom algoritme časová zložitosť závisí od používanej dátovej štruktúry s ohodnotením hrán. Pre maticu susedností je to ***O(n^2).*** Časová zložitosť pri Kruskalovom algoritme je zložitosť ***O(E log E),*** kde E je počet hrán v grafe.

1.1.4 Prehľadávanie A* (A star algorithm)

Algoritmus A* je používaný na hľadanie optimálnych ciest v ohodnotenom grafe. A* používa princíp greedy algoritmu na nájdenie optimálnej cesty z daného začiatočného vrcholu do požadovaného cieľového vrcholu. Pod optimálnou cestou rozumieme najkratšiu, najrýchlejšiu alebo najlacnejšiu cestu. Na hľadanie tejto cesty sa používa heuristická funkcia, ktorá vracia približnú cenu cesty od aktuálneho vrcholu po cieľový vrchol. Ak by sa stalo, že cesta po ktorej algoritmus práve ide má väčšiu cenu, tak sa vráti na najmenšiu

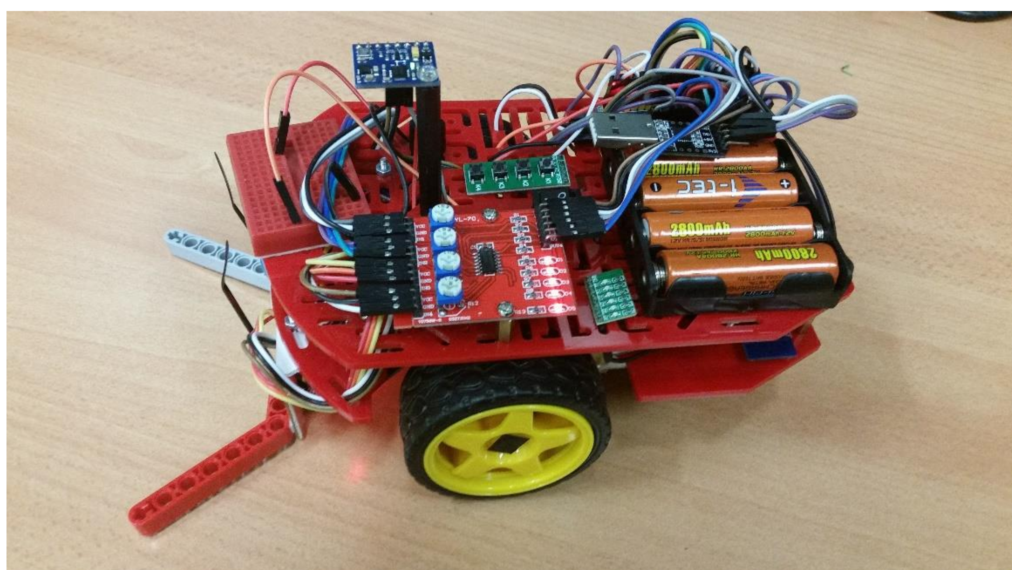
a pokračuje po tejto ceste kým nenájde cieľový vrchol. Vďaka veľkej presnosti v hľadaní riešení je táto metóda veľmi rozšírená. Heuristická funkcia v tejto metóde prehľadávania musí byť prípustná (admissible), čo znamená, že nebude vracať hodnotu väčšiu ako v skutočnosti je. Vďaka tomu je algoritmus optimálny.

Časová zložitosť algoritmu záleží od heuristiky, v najhoršom prípade stúpa počet navštívených vrcholov exponenciálne s dĺžkou cesty.

1.2 Tatrabot

V tejto podkapitole sa budem venovať technickým parametrom Tatrabota, popíšem dôležité črty jedno-čipového počítača, ktorým je robot Tatrabot riadený. Ďalej popíšem jeho senzorickú výbavu a nakoniec spomeniem software Tatrabota.

Tatrabot je robot slúžiaci ako učebná pomôcka pre študentov. Bol využitý predovšetkým pre cvičenia kurzu Princípy počítačov - hardvér, ktorý je nasadený v 1. ročníku v bakalárskom študijnom programe Aplikovaná informatika, kde sa na ňom študenti učia základy programovania v jazyku C, základy spracovania signálov a riadenia[9]. Využíva sa na implementovanie a vyskúšanie robotických algoritmov. Študenti a nadšenci si môžu vyskúšať rôzne algoritmy, od základných ako napríklad algoritmus na sledovanie čiary, cez zložitejšie algoritmy ako pohyb robota po bludisku až po pokročilé algoritmy umelej inteligencie.

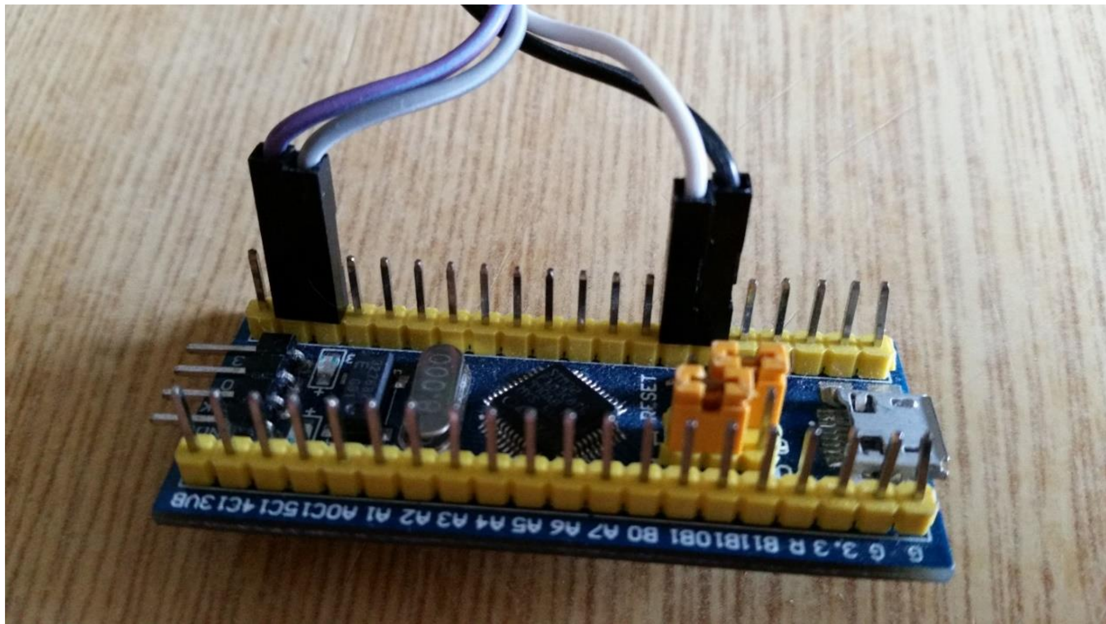


Obr. 1 Fotografia samotného Tatrabota

Tatrabot je riadený 32-bitovým jedno-čipovým počítačom rady STM32.

Vlastnosti počítačovej rodiny STM32:

- **Výpočtové jadro (CPU)** - ARM 32-bit Cortex M3 s maximálnou frekvenciou 72 MHz
- **Pamäť** – 64 alebo 128 kilobajtov pamäte typu flash,
    - o 20 kilobajtov SRAM (Static Random Access Memory) - uchováva údaje v registroch z preklápacích obvodov



Obr.2 Základná doska STM32

Robot má taktiež bohatú senzorickú výbavu: 3 osí akcelerometer, gyroskop, magnetometer, barometer, teplomer, 2 IR senzory na detekciu prekážok, SHARP IR senzor na meranie vzdialenosti, 4-kanálový senzor na čiaru, zvukový výstup, tlačidlá, signálne LED, otáčkové senzory, BlueTooth.

Čo sa týka softwaru Tatrabota, tak obsahuje firmware napísaný v jazyku C, ktorý tvorí základný framework, na ktorom môžu študenti stavať vlastné aplikácie.

Ďalej môžeme vidieť tabuľku parametrov Tatrabota.

Parametre Tatrabota:

Konštrukcia robota	2WD Beginner Robot Chassis V2
Rozmery robota	175mm x 109mm
Rozmery kolies (priemer x šírka)	67mm x 26mm
Hmotnosť bez zdroja napájania	465g
Hmotnosť so zdrojom napájania	561g
Napájanie	4x NiMH 2800mAh AA batérie
Pohon	2x Dagu robot DG01D Mini DC Gearbox
Rýchlosť motorov pri voľnobehu	90 RPM ± 10
Základná doska	STM32F103C8T6 ARM STM32
Procesor	ARM 32-bit Cortex M3
Senzorická výbava	3-osí akcelerometer, gyroskop, magnetometer, barometer, teplomer, 2 IR senzory na detekciu prekážok, SHARP IR senzor na meranie vzdialenosti, 4-kanálový senzor na čiaru, otáčkové senzory
Ďalšie pripojiteľné senzory	kompas
Ostatné zariadenia	vypínač, tlačidlá, signálne LED diódy, BlueTooth.
Programovací jazyk	C

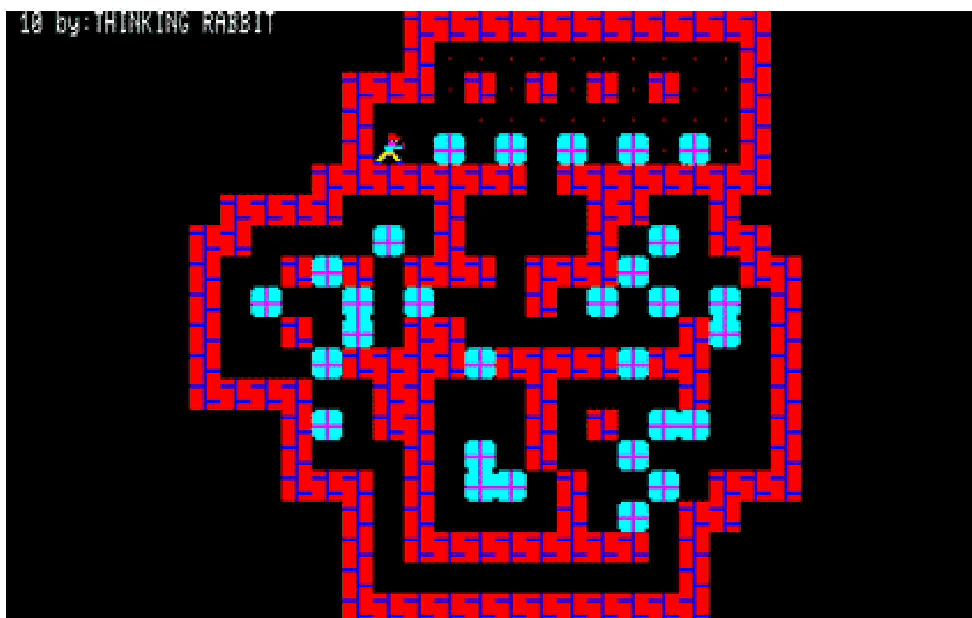
1.3 Sokoban

Informácie v nasledujúcej kapitole sú použité zo zdroja [8].

Sokoban (skladník) je logický problém, väčšinou reprezentovaný ako počítačová hra, v ktorej hráč ovláda postavu skladníka, ktorý sa pohybuje v dvojrozmernom labyrinte, kde sa nachádzajú krabice a niektoré políčka sú určené ako cieľ. Cieľom hráča je presúvať krabice v levely tak, aby všetky krabice skončili na políčku označenom ako cieľ. Labyrint hry je zobrazený pohľadom zhora. Táto logická hra je väčšinou implementovaná ako videohra.

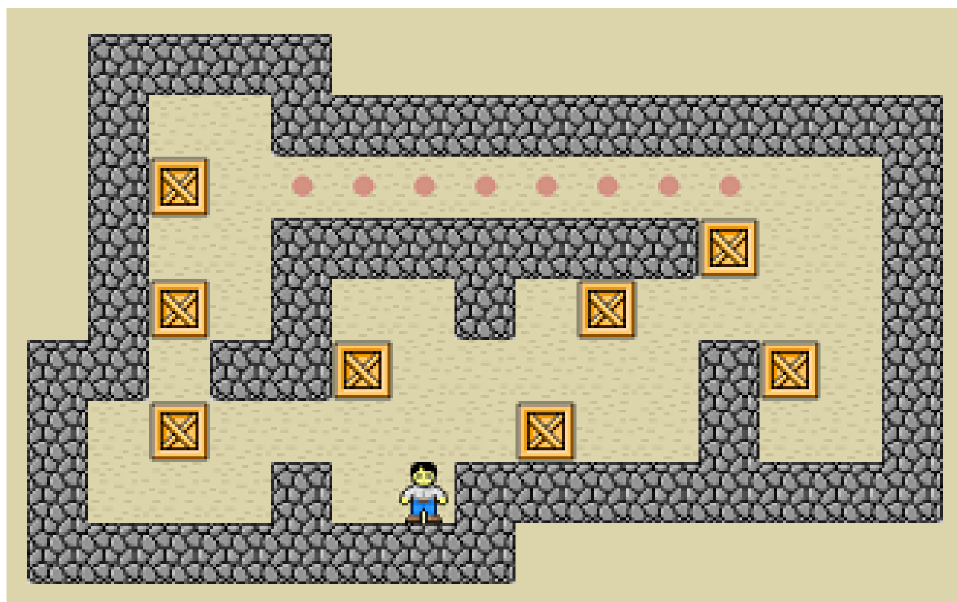
Hru Sokoban prvýkrát vymyslel Hiroyuki Imabayashi v roku 1981 v rámci jednej súťaže skladov o vymyslenie motivačnej hry pre zamestnancov. Následne v roku 1982 bola táto hra prvýkrát vydaná japonskou softwarovou firmou Thinking Rabbit. Prvé vydanie hry Sokoban obsahovalo dvadsať levelov a v priebehu nasledujúcich rokov vydala tá istá firma štyri ďalšie verzie tejto hry, ktoré obsahujú viac ako 850 levelov. Najnovšia verzia hry Sokoban vydaná firmou Thinking Rabbit bola vydaná v roku 2015 a obsahuje 200 levelov.

Pravidlá hry sú veľmi jednoduché. Skladník sa môže hýbať len v štyroch smeroch : vľavo, vpravo, hore a dolu. Keď skladník narazí na krabicu, môže ju len tlačiť, nie ťahať a to tiež len v štyroch smeroch, ktorými sa sám môže pohybovať. Súčasne môže skladník pracovať len s jednou krabicou. Cieľom hry je nielen dostať všetky krabice na určené cieľové políčka, ale zároveň to dosiahnuť na čo najmenší počet ťahov.



Obr.3 Ukážka levelu hry Sokoban z roku 1982

(http://sokoban-jd.blogspot.sk/2012/10/sokoban-30th-anniversary-1982-2012.html)



Obr. 4 Ukážka levelu novšej verzie hry z roku 1996

(http://sokoban-jd.blogspot.sk/2015/09/sokoban-lessons-lines-5-8-boxes.html)

Obr. 5 Ukážka 3D levelu hry Sokoban na platforme Android

(https://play.google.com/store/apps/details?id=com.keygames.sokobanunited)

Dnes existuje veľa rôznych verzií tejto hry. V niektorých najnovších verziách môže skladník krabice dokonca aj ťahať nie len tlačiť. Niektoré ďalšie verzie hry majú označené krabice rôznymi farbami a treba ich uložiť na cieľové políčka príslušnej farby. Od prvého vydania Sokobana, vyšli mnohé ďalšie hry na rovnakom princípe skoro na všetkých platformách od PC až po konzoly a mobilné telefóny a iné.

1.4 Sokoban solver

Najznámejším programom na riešenie Sokobana je Rolling Stone, ktorý bol vytvorený na University of Alberta v provincii Alberta v Edmontone, Kanada.

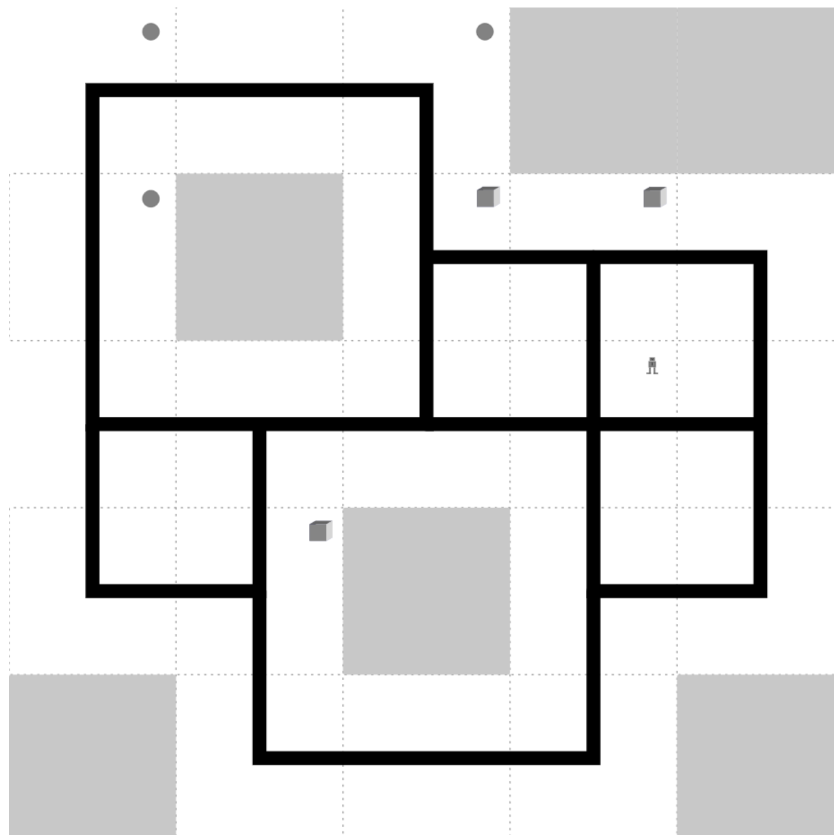
Na prehľadávanie bol použitý algoritmus IDA* (Iterative deepening A*).
Tento algoritmus IDA* je cyklicky sa prehlbujúce hľadanie algoritmom A*. Každé opakovanie bude obmedzeným hľadaním do hĺbky – namiesto ohraničenej hĺbky sa pracuje s hraničnou cenou, vygenerovanou algoritmom A* V jednom opakovaní sa rozvíjajú všetky uzly, ktorých hĺbky nepresiahnu vygenerovanú hraničnú hodnotu. K tomuto algoritmu ešte pridali ďalšie vylepšenia a vďaka nim vie Rolling Stone vyriešiť všetkých 90 levelov použitých v prvom vydaní Sokobana. [7]

Existuje ešte aj mnoho ďalších solverov, ktoré sú určené aj pre iné platformy ako PC. Žiadny z nich sa však nevyrovná Rolling Stone. Väčšina z nich vie vyriešiť približne 80 pôvodných levelov.

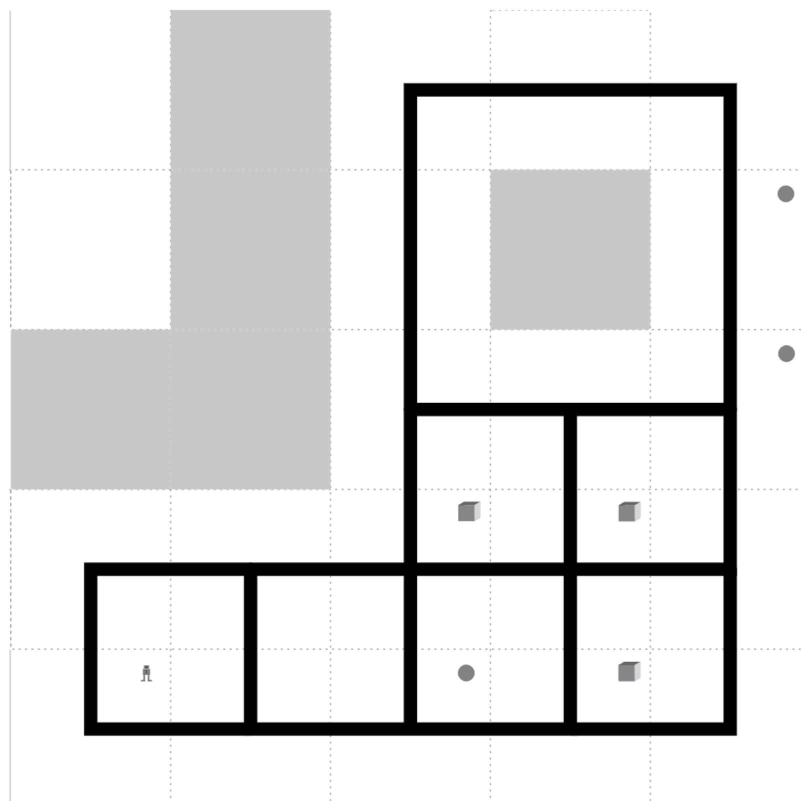
2 Návrh riešenia

V nasledujúcej kapitole opíšem návrh riešenia hry Sokoban, popíšem používané dátové štruktúry. Taktiež definujeme používané funkcie a ich funkcionalitu.

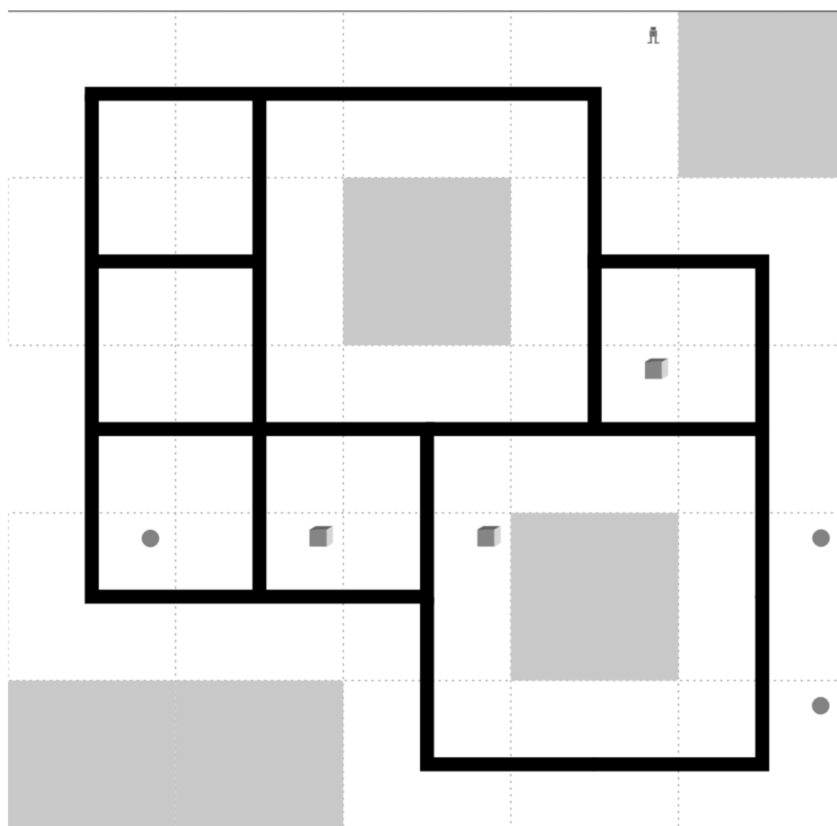
Ako som spomenul v kapitole 1.4 v hre Sokoban sa skladník pohybuje po mape a ukladá krabice na cieľové pozície. K mojej práci budeme používať 4 mapy rozmerov 5x5 políčok, po ktorých sa bude Tatrabot pohybovať. Mapa obsahuje cieľové políčka (krúžok), krabice(kocka), robota (panáčik) a steny (šedé políčka). Čierne čiary sú vodiace čiary, ktoré bude Tatrabot vyžívať pri pohybe v labyrinte.



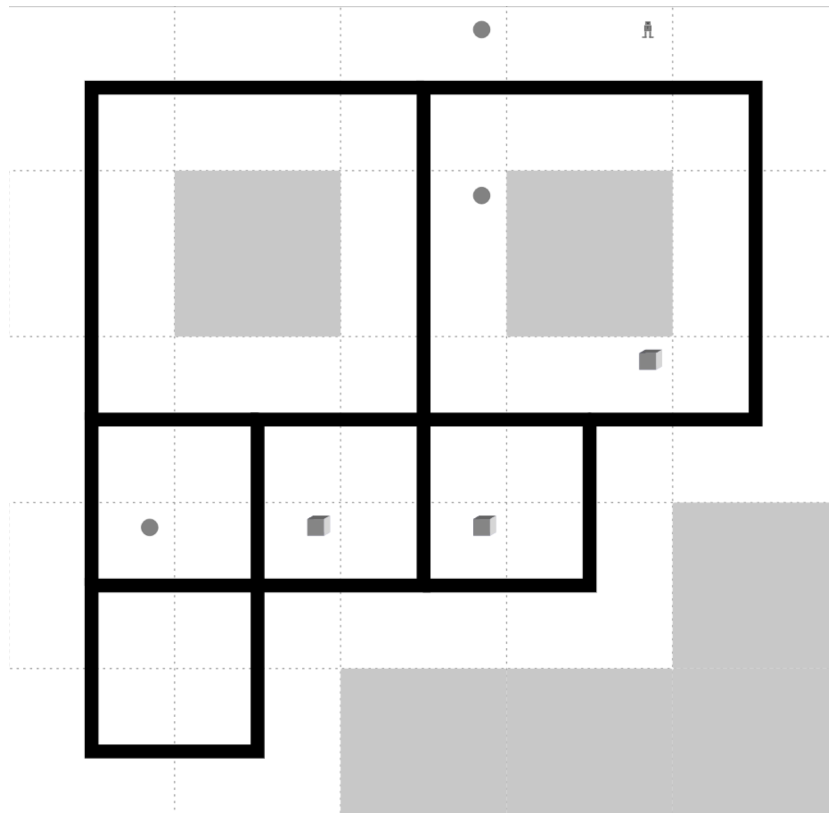
Obr. 6 Ukážka mapy č.1



Obr. 7 Ukážka mapy č.2



Obr. 8 Ukážka mapy č.3



Obr. 9 Ukážka mapy č.4

Fyzické mapy na obrázkoch vyššie som si následne zakódoval do konštánt (typu string), ktoré bude program používať pri prehľadávaní.

2.1 Prvá verzia programu v C++

Na prehľadávanie vytvorenej mapy som sa rozhodol použiť informovanú metódu prehľadávania algoritmom A*.

Zo stringu mapy sa vyrobí počiatočný stav hry, v ktorom sa získajú pozície všetkých cieľových políčok, pozícii krabíc a pozícia robota, ktoré sa následne uložia do dynamického poľa (typu std::vector). Ďalej sa v tomto počiatočnom stave vytvoria premenné pre štatistiku, v ktorých sa bude pamätať zoznam krokov, počet krokov, celková cena cesty a štatistika celkového počtu vrcholov, počet otvorených, zatvorených vrcholov, počet vygenerovaných duplikovaných vrcholov atď. Následne sa vygenerujú všetky možné stavy, ktoré môžu nastať pri pohybe robota po mape. Od pozície robota existujú vždy 4 možnosti na pohyb: hore, dolu, vľavo, vpravo v prípade že sa na jednej z týchto pozícii nenachádza stena.

Pri programovaní funkcie na generovanie nasledujúcich stavov využijem first-in first-out (FIFO) front (std::queue), v ktorom platí, že prvý prvok, ktorý sa do queue vloží sa prvý aj vyberie. Do tohto radu sa pri generovaní nových stavov vkladajú všetky platné stavy, ktoré ešte neboli vygenerované.

Následne algoritmus prehľadávania potom začne tým, že sa pre každý vygenerovaný stav spustí heuristická funkcia, ktorá ohodnocuje nové stavy využitím Manhattanskej vzdialenosti (počíta sa ako absolútna hodnota súčtu rozdielov súradníc bodov, pre ktoré sa využíva) a odhaduje zostávajúcu vzdialenosť k cieľovému políčku.

Vzorec pre Manhattanskú vzdialenosť:

$$d = \sum_{i=1}^{n} |x_i - y_i|$$

Kde x, y sú súradnice bodov, pre ktoré sa vzdialenosť počíta.

Najskôr sa odhad vypočíta pre vzdialenosť robota k najbližšej dosiahnuteľnej pozícii krabice a následne sa pre každú krabicu počíta vzdialenosť ku každému prázdnemu cieľovému políčku. Keďže nie každá kocka sa musí dostať k jej najbližšiemu cieľu tak skutočný počet prejdených krokov bude buď menší alebo rovnaký alebo väčší ako číslo ohodnotenia z heuristickej funkcie, vďaka čomu je táto heuristika prípustná a môžeme ju použiť. Vzdialenosť sa potom pripočíta k výslednému skóre daného stavu. Čím menšie je skóre stavu tým lepšie a tým väčšia je priorita daného stavu.

Pri generovaní stavov sa pridáva do zoznamu krokov pre aktuálny stav reťazec prejdených krokov, ktorý sa na konci zobrazí ako postupnosť krokov, ktoré musel robot vykonať kým neprišiel do cieľového stavu. Po každom kroku sa aktualizuje v danom stave mapa, ktorá má údaje o aktuálnych polohách krabíc, robota a cieľov.

Následne sa na vygenerované stavy spustí algoritmus prehľadávania A*. V samotnom prehľadávaní sa využíva dátová štruktúra obojsmerného radu (std::deque - **d**ouble-**e**nded **que**ue) do ktorej sa pridávajú vygenerované otvorené stavy. Následne sa v obojsmernom fronte prehľadajú všetky obsiahnuté stavy, porovnávajú sa ich heuristické ohodnotenia a ak aktuálne vybraný stav má väčšiu prioritu ako ostatné vygenerované stavy nachádzajúce sa v OPEN zaradí sa späť do deque na prvú pozíciu, čím nám vlastne vznikne prioritný rad s najlepšími stavmi na začiatku.

Ako prvé sa prehľadávajú stavy s najvyššou prioritou z heuristickej funkcie, teda stavy, ktoré dostali najnižšie ohodnotenie od heuristickej funkcie. Algoritmus prehľadáva postupne všetky vygenerované stavy až kým nedospeje do cieľového stavu. Funkcia na

zistenie, či je stav cieľový skontroluje aktuálnu mapu v danom stave a zisťuje, či sa v mape nachádzajú ešte voľné cieľové políčka a krabice, ktoré ešte nie sú v cieli. Zároveň aj kontroluje, či robot náhodou nestojí na cieľovom políčku. Ak sú splnené všetky podmienky cieľového stavu tak sa stav označí ako finálny a algoritmus končí prehľadávanie.

Tým, že sa do platných stavov budú vkladať len tie, ktoré neboli duplicitné sa značne zníži časová a pamäťová náročnosť, pretože sa znížil počet prehľadávaných vrcholov. Tento postup riešenia bude pravdepodobne dostačujúci na to aby sa zmestil do operačnej pamäte Tatrabota.

2.2 Druhá verzia programu v C

Keďže Tatrabot má len 20 kB operačnej pamäte a pri implementácii algoritmu popísaného v kapitole 2.1 sme zistili, že prvé riešenie algoritmu bolo časovo a pamäťovo náročné kvôli tomu, že ku každému stavu sa pripájala aj aktuálna mapa stavu a vždy znova sa prehľadávala pri generovaní nových stavov. Toto riešenie Tatrabot nedokázal zmestiť do operačnej pamäte, preto sme sa rozhodli navrhnúť iné riešenie, v ktorom budeme dostupnú pamäť využívať na maximum.

V druhej verzii programu na nájdenie cesty pre robota z miesta [start_r, start_s] na miesto [cieľ_r, cieľ_s] navrhneme funkciu, ktorá pre dané súradnice pomocou algoritmu A* zistí či cesta existuje.

Funkcia bude využívať nasledovné dátové štruktúry:

- **open[ ]** – zoznam otvorených vrcholov kandidujúcich na navštívenie v prehľadávanom priestore – zoznam je vhodné implementovať prioritným frontom, kde sa pri usporiadaní využije ohodnotenie podľa heuristickej funkcie $g(x) = h(x) + dist(root, x)$
- **closed[ ]** – zoznam vrcholov, ktoré už boli počas prehľadávania navštívené
- **arrived_from[ ]** – identifikátor vrchola, z ktorého bol každý vrchol v **open[ ]** aj v **closed[ ]** objavený najkratšou cestou (podľa algoritmu A*)
- vrchol tvorí pozícia robota (rr, rs)

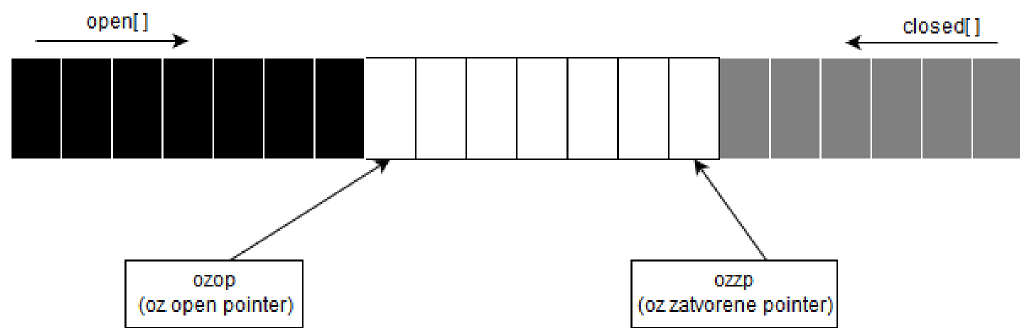
Ďalej popíšem pseudokód algoritmu (podľa A*), ktorý budem používať na nájdenie cesty pre robota z miesta [start_r, start_s] na miesto [cieľ_r, cieľ_s]

1. Do zoznamu open[ ] pridaj [start_r, start_s]
2. Opakuj, kým open[ ] nie je prázdny:
   2.1. vyber z open [ ] pozíciu [r, s] s minimálnym g(x)
   2.2. odstráň [r, s] z open[ ]
   2.3. do open[ ] pridaj všetky vrcholy (pozície) dosiahnuteľné z [r, s] – 4 susedné políčka, na ktorých sa v mape nenachádza prekážka (stena/krabica) a zároveň zaznamenaj v arrived_from [ ] pre tieto vrcholy vrchol [r, s],], ale len pre tie, ktoré nie sú v open[ ] ani v closed[ ], ale ak sú v open[ ] s väčšou dist(root, x) ako aktuálne (dist(root, x)+1), tak im prepíšem dist[y] aj arrived_from[y]
   2.4. ak je ktorýkoľvek z nasledovníkov zhodný s pozíciou [ciel_r, ciel_s] tak preruš cyklus 2 s úspechom
3. ak bol cyklus prerušený s úspechom, tak zrekonštruuj cestu:
   3.1. začni vo vrchole [cieľ_r, cieľ_s] → [r, s], vynuluj cestu cesta[ ]
   3.2. kým [r, s] je iný ako [start_r, start_s]
        pridaj do cesty vrchol [r, s]
        [r, s]← arrived_from[r, s]
   3.3. pridaj do cesty vrchol [start_r, start_s]
   3.4. obráť cestu (reverse)
4. ak bol cyklus prerušený neúspešne, cesta neexistuje

Ďalej popíšem návrh samotného veľkého prehľadávacieho algoritmu.

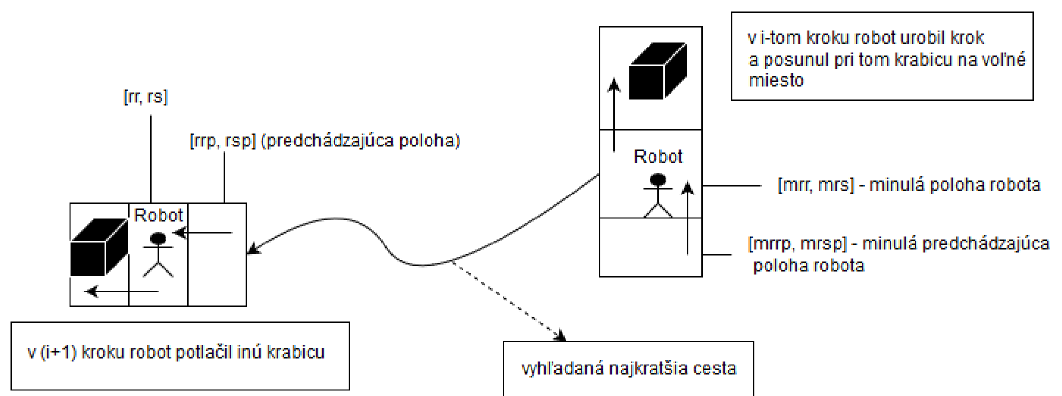
Analogicky ako vyhľadávanie cesty pomocou algoritmu A*, ale stavmi v tomto prípade budú konfigurácie polôh krabíc na mape a poloha robota.
Nový stav teda bude mať štruktúru[kr_1, ks_1, kr_2, ks_2, kr_3, ks_3, rr, rs], pričom za rovnaké považujeme také dva stavy, v ktorých sa nezmenila žiadna pozícia krabice iba poloha robota, pretože takýto stav nemá vplyv na priebeh hry.
V tomto prípade však nastáva problém, keďže nevieme dobre odhadnúť veľkosť polí open[ ] a closed[ ], pretože veľkosť nami prehľadávaného priestoru závisí od konkrétnej mapy a je potenciálne obrovská. Pretože by sme chceli využiť všetku zostávajúcu pamäť, ale nevieme aký bude vhodný pomer medzi počtom prvkov open[ ] a closed[ ]. Preto sme navrhli postup, pomocou ktorého umiestnime do jedného poľa aj prvky open[ ] a prvky closed[ ] , pričom

vrcholy open [ ] sa ukladajú od začiatku poľa a vrcholy closed[ ] sa ukladajú od konca toho istého poľa. Časť poľa, pri ktorej prechádzame open vrcholy využíva na prechádzanie open pointer a na prechádzanie využívame closed pointer. Týmto spôsobom využijeme pamäť maximálne efektívne.



Obr. 10 Návrh spoločného poľa pre otvorené a zatvorené vrcholy

V našom návrhu vonkajší algoritmus prehľadávania A* volá vnútorný algoritmus prehľadávania A* a to keď nastane situácia zachytená na Obr. 12.



Obr. 11 Situácia volania vnútorného algoritmu prehľadávania

2.3 Návrh pohybu robota po mape

V nasledujúcej kapitole budem popisovať návrh pohybu robota.

Pri riešený pohybu robota po labyrinte budeme používať knižnicu **tatrabot.h** v ktorej sú naprogramované základné inštrukcie pre samotného robota. Tatrabot najskôr zavolá funkciu sokoban( ), ktorú sme implementovali v kapitole 3.2. Po prebehnutí algoritmu prehľadávania A* na základnej doske STM32 funkcia vráti robotovi zrekonštruovanú cestu, ktorá nastala keď sa objavil cieľový stav. Keďže reprezentácia cesty bola vygenerovaná v reťazci, budeme musieť tento reťazec po znakoch prechádzať aby sme zistili, ktorým smerom sa má robot pohnúť. Následne nám funkcia **decode_move( )** dekóduje smer pohybu zo stringu na novy_smer, ktorým sa treba pohybovať. Začína smerom na sever.
Algoritmus pohybu bude po dekódovaní smeru nasledovný:

- Ak sa smer pohybu nemení – choď dopredu ( )
- ak sa mení smer pohybu vľavo/vpravo – zatoč vľavo/vpravo, choď dopredu
- ak sa mení smer pohybu na opačný – otoč sa o 180°, choď dopredu

Počas pohybu je dôležité, aby robot sledoval vodiace čiary na mape, aby nám nenastala situácia, že nám robot pri pohybe odíde z ihriska. Po vykonaní pohybu a dosiahnutí cieľového stavu by mal robot napríklad zvukovým signálom oznámiť, skončil vykonávanie pohybu.

3 Implementácia

V nasledujúcej kapitole budem popisovať implementáciu s použitím algoritmu, ktorý som opísal v kapitolách 1.2 a 1.2.4 a tiež spomeniem zaujímavé problémy, ktoré som musel počas implementácie riešiť.

Aplikáciu vyvíjam vo vývojovom prostredí MS Visual Studio 2015 a ChibiOS. ChibiOS je rýchly a kompaktný RTOS (real-time operating system), ktorý podporuje rôzne architektúry a demo aplikácie pre mikroprocesory rodiny STM32 a iné.

3.1 Prvá verzia implementácie algoritmu prehľadávania v C++

Podľa návrhu programu spomenutého v kapitole 2.1 sme implementovali prehľadávanie nasledovne.

Na reprezentáciu stavu hry sa vytvorí dátová štruktúra **struct Stav**, ktorá obsahuje nasledovné atribúty:

- std::string stav_string – obsahuje aktuálnu reprezentáciu stavu v podobe reťazca
- std::string zoznam_krokov – reťazec prejdených krokov
- int hlbka – číslo ako hlboko algoritmus prišiel pri prehľadávaní
- int kroky – celkový počet vykonaných krokov
- int potlacenia – celkový počet potlačení krabíc
- int celkova_cena – celková cena cesty
- int heuristika_hodnotenie – heuristické ohodnotenie pre daný stav

Ku structu Stav sa vytvorí ešte **struct Statistika,** ktorý počíta informácie algoritmu:

- int pocet_vrcholov – celkový počet vygenerovaných stavov
- int pocet_opakovanych – počet vygenerovaných duplicitných stavov
- int pocet_otvorenych_vrcholov – počet open [ ] vrcholov
- int pocet_prehladanych_vrcholov – počet closed [ ] vrcholov

Z reťazca mapy sa vyrobí počiatočný stav hry, v ktorom sa získajú pozície všetkých cieľových políčok, pozícii krabíc a pozícia robota, ktoré sa následne uložia do dynamického poľa krabice[ ], ciele [ ], robot[ ].

Hľadanie všetkých nasledovníkov realizuje funkcia **generuj_nasledujuce_stavy,** ktorá pre každý pohyb robota vygeneruje nový stav pre kroky v štyroch smeroch. Funkcia

zisťuje na aké políčko bol vykonaný krok, podľa toho, ktorým smerom sa chce robot pohnúť. V prípade, že sa chce robot pohnúť na políčko, na ktorom nie je prekážka (prekážka je krabica/stena) sa vykoná krok a nastaví sa nová mapa levelu. V prípade, že je v smere pohybu stena tak vyskočí z cyklu a v prípade, že je tam krabica tak sa nastavia aj nové súradnice krabice. Štruktúra je nasledovná:

- **krok hore**
    - **prípad prázdneho políčka** – vykoná sa krok, nastaví sa nová mapa levelu
    - **prípad cieľového políčka** - vykoná sa krok, nastaví sa nová mapa
    - **prípad políčka s krabicou** – vykoná sa krok, nastavia sa nové súradnice krabice na súradnice políčka, ktoré nasleduje v smere pohybu za krabicou. Ak je tam prekážka tak vyskočí z cyklu, pretože sa krabicou nedá v danom smere pohnúť
    - **prípad políčka s krabicou v cieľovom políčku** – analogicky ako v prípade políčka s krabicou

    aktualizovanie štatistiky pre daný stav
- **krok vpravo**
    - **prípad prázdneho políčka** – vykoná sa krok, nastaví sa nová mapa levelu
    - **prípad cieľového políčka** - vykoná sa krok, nastaví sa nová mapa
    - **prípad políčka s krabicou** – vykoná sa krok, nastavia sa nové súradnice krabice na súradnice políčka, ktoré nasleduje v smere pohybu za krabicou. Ak je tam prekážka tak vyskočí z cyklu, pretože sa krabicou nedá v danom smere pohnúť
    - **prípad políčka s krabicou v cieľovom políčku** – analogicky ako v prípade políčka s krabicou

    aktualizovanie štatistiky pre daný stav

- **krok dolu**
    - **prípad prázdneho políčka** – vykoná sa krok, nastaví sa nová mapa levelu
    - **prípad cieľového políčka** - vykoná sa krok, nastaví sa nová mapa
    - **prípad políčka s krabicou** – vykoná sa krok, nastavia sa nové súradnice krabice na súradnice políčka, ktoré nasleduje v smere pohybu za krabicou. Ak je tam prekážka tak vyskočí z cyklu, pretože sa krabicou nedá v danom smere pohnúť
    - **prípad políčka s krabicou v cieľovom políčku** – analogicky ako v prípade políčka s krabicou

    aktualizovanie štatistiky pre daný stav
- **krok vľavo**
    - **prípad prázdneho políčka** – vykoná sa krok, nastaví sa nová mapa levelu
    - **prípad cieľového políčka** - vykoná sa krok, nastaví sa nová mapa
    - **prípad políčka s krabicou** – vykoná sa krok, nastavia sa nové súradnice krabice na súradnice políčka, ktoré nasleduje v smere pohybu za krabicou. Ak je tam prekážka tak vyskočí z cyklu, pretože sa krabicou nedá v danom smere pohnúť
    - **prípad políčka s krabicou v cieľovom políčku** – analogicky ako v prípade políčka s krabicou

    aktualizovanie štatistiky pre daný stav

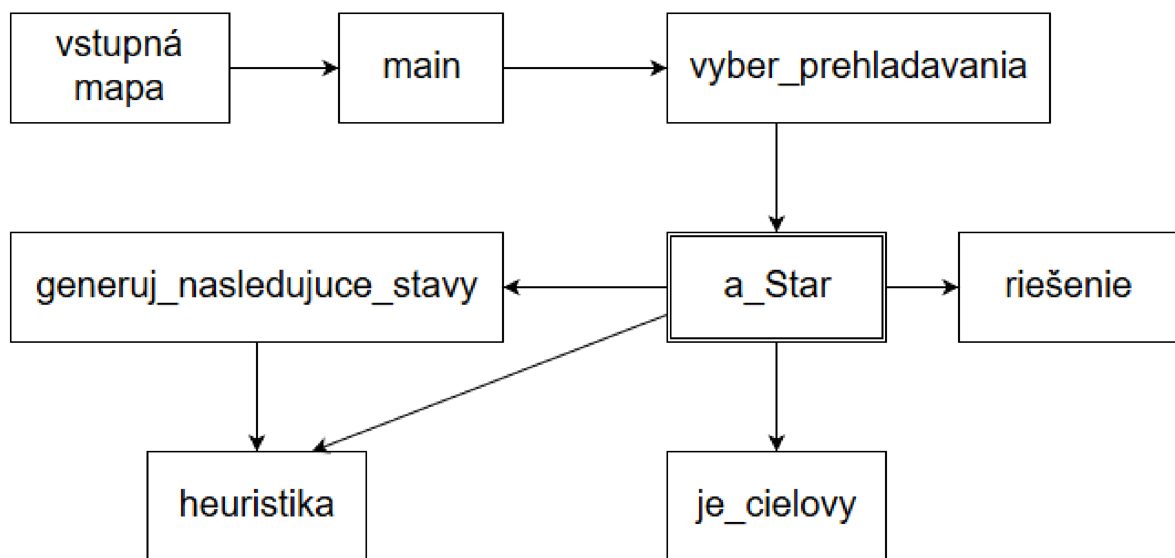
Pri generovaní krokov sa v každom stave aktualizujú premenné pre štatistiku, v ktorých sa bude pamätať zoznam krokov, počet krokov, celková cena cesty a štatistika celkového počtu vrcholov, počet otvorených, zatvorených vrcholov, počet vygenerovaných duplikovaných vrcholov a heuristické ohodnotenie stavu.

Stavy, ktoré vygenerujú funkciou popísanou vyššie následne dostane funkcia **a_Star**, ktorá aktuálne vygenerovaný stav vloží do radu platne_stavy. Následne v cykle kým nie sú platne_stavy prázdne kontroluje či nie je aktuálny platný stav už vložený v open [ ], či nie je v closed [ ] a či nie je duplikátny. Keď splní podmienku tak sa aktuálny platný stav pridá do **deque** na prioritnú pozíciu podľa heuristického hodnotenia. Vrchol s najvyššou prioritou je na prvej pozícii.

Funkcia a_Star zároveň pomocou funkcie **je_cielovy** overuje či aktuálny stav s najvyššou prioritou (najlepšie heuristické ohodnotenie) nie je cieľový.

Funkcia je_cielovy zistí či sa v danom stave nenachádzajú voľné ciele, voľné krabice a či robot nestojí na cieľovom políčku. Ak je táto podmienka splnená tak sa stav označí za finálny, prehľadávanie končí a vypíše sa cesta spolu zo štatistikou, ktorá získala pri generovaní nasledujúcich ťahov.

Na nasledujúcom obrázku je znázornené, ktorá funkcia v našom programe volá ktorú. Dvojitý obdĺžnik označuje hlavnú funkciu, ktorá hľadá riešenie.



Obr. 12 Znázornenie volaní funkcií v programe

Pri testovaní vyššie popísanej implementácie sme narazili na viacero problémov.

Prvým hlavným problémom bolo, že toto riešenie v jazyku C++ bolo veľmi pamäťovo náročné, pretože pri každom stave sa zapamätávala aktuálna mapa, ktorá sa vždy musela znova prehľadať (vysoká časová náročnosť).

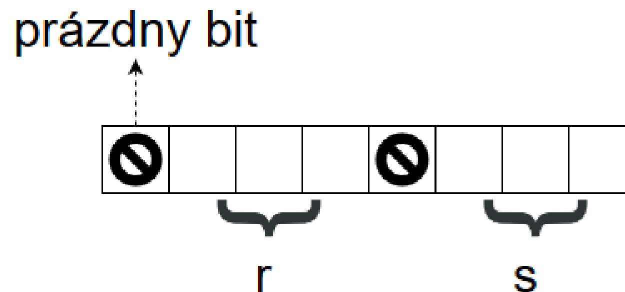
Ďalším problémom bolo, že pri riešení sa využívali dynamické vectory, ktoré nemali fixnú veľkosť, čo spôsobilo problém kvôli veľkosti operačnej pamäte Tatrabota, ktorá má len 20 kB.

Posledným problémom bolo, že keď sme chceli implementáciu algoritmu v C++ nasadiť na Tatrabota, ktorý si mal sám cestu vypočítať a vykonať, potrebovali sme pri programovaní základnej dosky STM32 využiť ChibiStudio, v ktorom sa táto doska programuje. Pri linkovaní riešenia algoritmu v C++ spolu s kódom v C, ktorý využíva Tatrabot na svoje inštrukcie sme dostali chybu, ktorú sme nevedeli odstrániť. Chyba bola obsiahnutá v samotnom ChibiStudiu, kde bola len experimentálna verzia C++ inštrukcií pre STM32, pri ktorej nebolo zaručené fungovanie externého C++ kódu v samotnom STM32.

Kvôli vyššie popísaným problémom sme sa rozhodli navrhnúť a implementovať nové riešenie v jazyku C, v ktorom sa budeme snažiť maximálne šetriť dostupnou pamäťou.

3.2 Druhá verzia implementácie algoritmu prehľadávania v C

V implementácii hľadania cesty budeme kvôli maximálnemu využitiu pamäte vrcholy [r, s] reprezentovať 1 bajtom. Horné 4 bity budú zodpovedať súradnici r a dolné štyri bity budú obsahovať súradnicu s. Maximálne rozmery ihriska teda môžu byť do veľkosti 16x16.



Obr. 13 Reprezentácia vrcholu jedným bajtom

Jednotlivé smerový vektory budeme reprezentovať poľami:

delta_r[4] = { -1, 0, 1, 0 }

delta_s[4] = { 0, 1, 0, -1 }

Polia open[ ] a closed[ ] môžu obsahovať každé pole ihriska najviac raz a teda pre naše mapy rozmerov 5x5 postačuje pole fixnej dĺžky 25, pričom premenné:

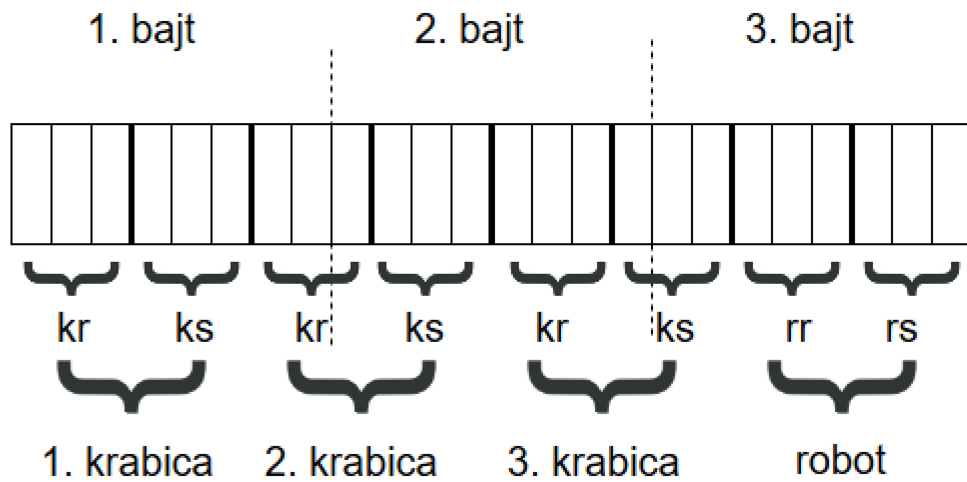
- **orp** (open read pointer) – označuje prvok poľa open[25] s najnižším indexom, ktorý je ešte využitý
- **owp** (open write pointer) – prvý voľný prvok

Operácia get( ) vráti prvok s najlepším ohodnotením na pozícii vybraný a následne skráti využitú dĺžku poľa tak, že presunie prvý prvok na pozíciu vybraného: open[orp++] → open[vybraný]

Hľadanie všetkých nasledovníkov realizuje funkcia, ktorá skúša pohnúť každou krabicou v každom smere a výsledné vrcholy odovzdá kvôli efektívnosti cez globálne pole nasledovníci[12] kde 12 je maximálny počet nasledovníkov pre 3 krabice v 4 smeroch. Premenná pocet_nasledovnikov určí počet nasledovníkov z daného stavu. Do poľa nasledovníci[pocet_nasledovnikov] sa uloží vygenerovaný nový stav.

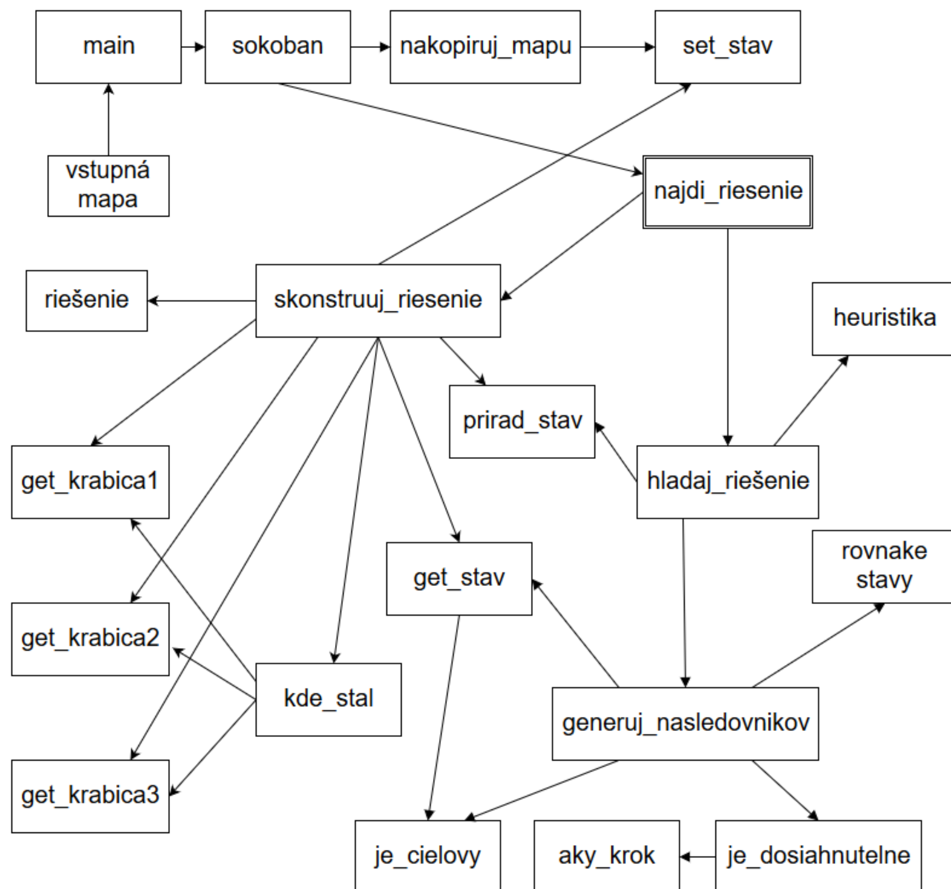
Heuristické ohodnotenie daného stavu k algoritmu A* počíta funkcia **heuristika,** ktorá využíva Manhattanskú vzdialenosť medzi robotom a najbližšou dosiahnuteľnou krabicou a medzi krabicami a cieľovými políčkami.

Pri implementácii veľkého prehľadávacieho algoritmu, musíme kvôli obmedzenej pamäti na STM32 použiť maximálne efektívnu reprezentáciu stavov pomocou 3 bajtov:



Obr. 14 Efektívna reprezentácia stavov pomocou 3 bajtov

Na nasledujúcom obrázku je znázornené, ktorá funkcia volá ktorú. Dvojitý obdĺžnik označuje hlavnú funkciu, ktorá hľadá riešenie.



Obr. 15 Znázornenie volaní funkcií v programe

3.3 Implementácia pohybu robota po labyrinte

Podľa návrhu programu spomenutého v kapitole 2.3 sme implementovali pohyb Tatrabota po labyrinte nasledovne:

- Pohyb dopredu nám zabezpečí funkcia **movingForward( )**, ktorá príkazom setMotor(3,SPEED) zapne oba motory a ich rýchlosť nastaví na konštantu SPEED
- Pohyb dozadu analogicky k pohybu dopredu vykoná funkcia **movingBackward( )**, ktorá príkazom setMotor(3, -SPEED) zapne oba motory avšak konštanta SPEED bude záporná, čo spôsobí pohyb vzad
- Zastavenie robota vykoná funkcia **stop( )**, ktorá príkazom setMotor(3, 0) nastaví obom motorom rýchlosť na 0, čo spôsobí zastavenie

Ďalej sme pri pohybe robota po mape potrebovali vyriešiť situáciu, keď robot príde na križovatku a potrebuje odbočiť. Odbáčanie na križovatkách nám zabezpečia nasledovné funkcie:

- **movingLeft( ) –** zabezpečuje pohyb vľavo tým, že zavolá príkazy
    - setMotor(2, SPEED) – rýchlosť na pravom motore je rovnaká
    - setMotor(1, 1000) – rýchlosť na ľavom motore je nižšia, čo spôsobí v tomto prípade zatočenie vľavo

- **movingRight( ) –** vykonáva pohyb vpravo tým, že zavolá príkazy
    - setMotor(1, SPEED) – rýchlosť na ľavom motore sa nemení
    - setMotor(2, 1000) – rýchlosť na pravom motore je znížená, čo spôsobí zatočenie doprava

- **rotateLeft ( ) –** funkcia vykonáva otočenie vľavo na mieste, volá príkazy
    - setMotor(2, TURN_SPEED)
    - setMotor(1, -TURN_SPEED)

    pravý motor teda ide dopredu konštantnou rýchlosťou a ľavý motor ide opačne

- **rotateRight( ) -** funkcia vykonáva otočenie vpravo na mieste, volá príkazy
    - setMotor(1, TURN_SPEED);
    - setMotor(2, -TURN_SPEED);

    V tomto prípade ľavý motor ide dopredu konštantnou rýchlosťou a pravý motor ide opačne

Pri našom pohybe ešte budeme potrebovať funkciu, ktorá bude sledovať čiary.
Funkcia **sledujCiaru( )** vykonáva nasledovné:

- movingForward( )
- kým hodnoty krajných senzorov na čiaru posielajú hodnotu logickej 0 (logická 0 znamená, že senzor vidí bielu farbu)
- ak hodnota niektorého z krajných senzorov je logická 1 (logická jednotka znamená že senzor vidí čiernu farbu) znamená to, že robot sa pri pohybe dostal niektorým z krajných senzorov na čiaru a musí sa vyrovnať. Podľa situácie sa zavolá buď funkcia movingLeft( ) – situácia keď ľavý senzor spozoroval čiaru, alebo sa zavolá funkciu movingRight( ) v situácii keď pravý senzor spozoruje čiaru

  Volanie funkcií movingLeft( )/movingRight( ) spôsobí, že sa robot vyrovná na čiaru. Je to potrebné kvôli tomu, že jedno koleso vždy ťahá rýchlejšie a tým sa robot vychyľuje z pôvodného smeru.

  K cyklu kontroly sme ešte pridali delay, aby Tatrabot stihol zaregistrovať hodnoty senzorov počas pohybu.

Ďalej sme s pomocou niektorých vyššie uvedených funkcií implementovali funkciu **dopredu(x)**, kde x je počet otáčok, ktoré majú motory vykonať v jednom kroku. V našom prípade jednému kroku (políčku na mape) zodpovedala hodnota 24 (asi 30 cm na mape). Funkcia dopredu(x) sleduje čiaru a zároveň vykonáva pohyb movingForward( ).

V prípade, že má robot v nasledujúcom kroku na niektorej križovatke odbočiť budeme potrebovať funkciu **otoc(smer)**, ktorá dostane ako parameter smer do ktorého chce robot ísť. Funkcia teda zavolá funkciu movingForward( ) a zároveň sleduje či náhodou neprišiel Tatrabot na križovatku (použijeme premennú detect_crossing). V prípade, že nastane daná situácia a zo smeru pohybu vieme, ktorým smerom sa treba v ďalšom kroku vydať nastavíme parameter funkcie na hodnotu ROT_LEFT alebo ROT_RIGHT podľa

nasledovného kroku kam sa chceme pohnúť. Robot potom začne zatáčať buď vľavo alebo vpravo. Pri kontrole zatáčania budeme zisťovať či robot v pohybe zatočenie znova senzormi na detekciu našiel čiaru. Ak ju našiel tak zatáčanie končí a robot sa pohne ešte kúsok dopredu(v našej situácii dopredu(15)) aby zostal na ďalšom políčku kam sa potreboval dostať a následne sa zavolá stop( ).

Väčší problém pri implementácii nastáva v situácii, kde sa robot potrebuje dostať na políčko za ním, pretože pri pohybe vzad nedokážeme sledovať čiaru. V takejto situáciu využijeme funkciu **otoc180(**):

- ktorá pošle robotovi príkaz movingBackward( ) aby sa posunul kúsok dozadu, čo je dôležité pri situácii, keď robot tlačil krabicu bez posunu vzad a vykonaní iba otočenia by krabicu posunul na pozíciu kde nemá byť
- funkcia potom zavolá rotateLeft( ) čo robota na mieste otáča až kým nenájde znova čiaru
- keď nájde čiaru zavolá sa stop( )

V situácii keď robot tlačil krabicu (funkcia **tlacil_krabicu(**)) a nepokračuje v jej tlačení v danom smere znova, ale ide vykonávať kroky iným smerom budeme ešte potrebovať funkciu **dotlacit(**). Použitie tejto funkcie je dôležité, pretože náš Tatrabot zastavuje nie v strede políčka, ale kúsok pred. Toto používame s toho dôvodu, že keby bola na danom políčku križovatka a robot by stál v nej a v nasledujúcom kroku by chcel odbočiť nevedel by zistiť, že na danej križovatke stojí a išiel by až po ďalšiu ktorú by našiel.
Funkcia dotlacit( ) teda vykoná to, že robot v takejto situácii krabicu posunie až na nasledujúce políčko a vráti sa pred križovatku príkazom movingBackward( ).

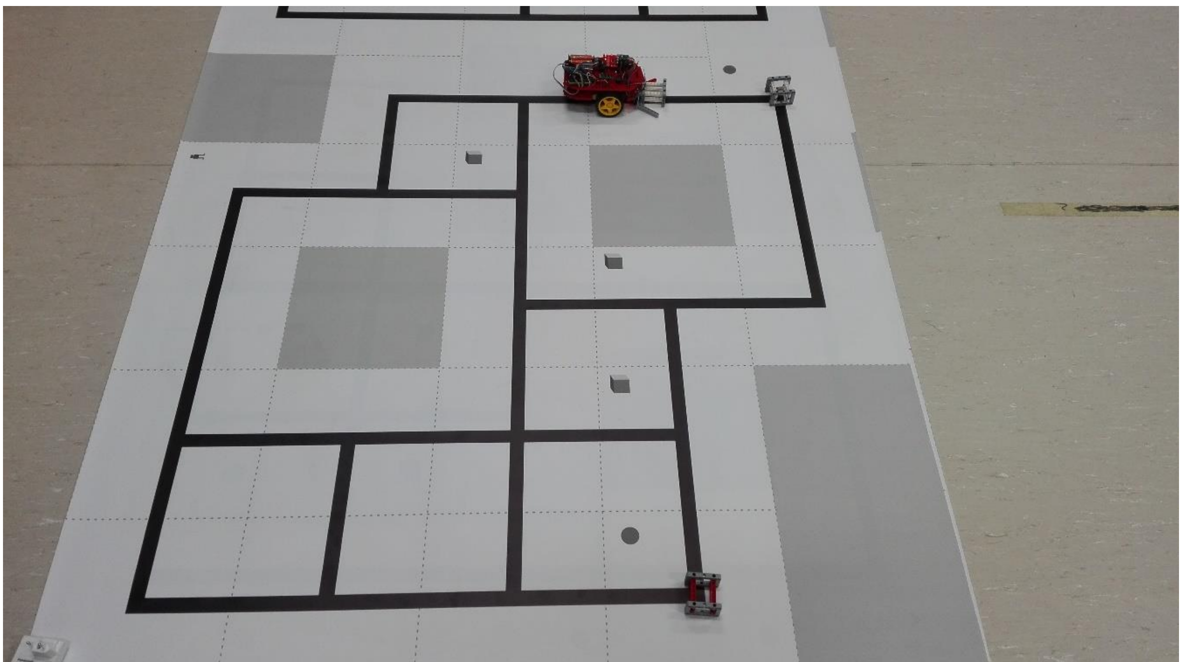
Potom sme ešte implementovali funkciu **vykonaj_riesenie**, ktorá vykoná dané pohyby podľa nájdenej cesty. Zavolá funkciu **decode_move(cesta)**, ktorá dostane ako parameter znak aktuálneho kroku a vráti nám novy_smer nasledovne:

- prípad 'd': novy_smer = 2; break;
- prípad 'u': novy_smer = 0; break;
- prípad 'l': novy_smer = 3; break;
- prípad 'r': novy_smer = 1; break;

Nakoniec sme ešte doprogramovali funkcie, ktoré nám vracajú zvukové výstupy z Tatrabota pre lepšie testovanie. Sú to funkcie:

- void oznam_smer(smer) – oznámi smer podľa decode_move( )
- void oznam_start_pohybu(void) – oznámi štart pohybu
- void oznam_koniec_pohybu(void) – oznámi koniec pohybu
- void oznam_zaciatok_dotlac(void) – oznámi začiatok dotlacit( )
- void oznam_koniec_dotlac(void) – oznámi koniec dotlacit( )
- void oznam_ciel(void) – oznámi, že nastal cieľový stav

Pri testovaní pohybu robota po labyrinte počas vykonávania cesty, bolo ešte potrebné ponastavovať konštanty, ktoré sa využívajú pri samotnom pohybe, aby bol pohyb čo najlepší a najpresnejší. V poslednej verzii, ktorú sa nám podarilo implementovať robot dokázal úspešne s malou pomocou prejsť našu fyzickú mapu a úspešne dosiahnuť cieľový stav. Na pohybe by sa ešte dali doladiť jednotlivé konštanty, ktoré sa využívajú, aby bol pohyb čo najlepší.



Obr. 16 Fotografia po úspešnom dosiahnutí cieľovej situácie na mape 3

Záver

Cieľom mojej bakalárskej práce bolo navrhnúť a implementovať vhodný algoritmus na prehľadávanie stavového priestoru pri hre Sokoban, oboznámenie sa s vlastnosťami a funkčnosťou samotného mobilného robotického systému Tatrabota a zároveň ich súčasne prepojenie. Následne som nadobudnuté poznatky využil v praxi pri implementovaní programu, ktorý bol schopný vyriešiť štyri rôzne levely hry Sokoban v simulácii na počítači, ako aj na robotickej platforme, ktorá dokázala na reálnej mape vyriešiť problém hry Sokoban a po úspešnej jazde dokázal Tatrabot umiestniť všetky krabice na ich cieľové políčka.

Zadanie mojej práce sa mi podarilo splniť, pričom som si osvojil základy umelej inteligencie a zároveň aj základy robotiky. Napriek tomu v prípade môjho ďalšieho záujmu o túto problematiku nemusí byť práca na túto tému ukončená, ale dá sa v nej pokračovať niekedy v budúcnosti. Pretože tu stále existuje priestor na zlepšovanie, aj čo sa týka efektívnosti algoritmu, poprípade porovnanie rôznych algoritmov ako aj zlepšenie samotného pohybu Tatrabota po labyrinte, pretože Tatrabot ako robotická platforma nie je dokonalý. Pre zlepšenie pohybu robota by bolo potrebné ešte spraviť nejaké úpravy aj na samotnom robotovi ako napríklad zvýšiť kapacitu zdroja jeho napájania a pridať regulátor, ktorý by zabezpečil stabilné napätie z batérii, pretože počas testovania sme museli niekoľkokrát vymieňať batérie, keďže pri poklese prúdu sa v niektorých situáciách robot nevládal dotočiť pri križovatkách a bolo mu potrebné trošku pomôcť. Existuje tu možnosť doladiť konštanty, ktoré sa využívajú pri pohybe, aby sa robot mohol pohybovať čo najpresnejšie. Popracovať by sa dalo aj na samotnom algoritme prehľadávania, v ktorom by sa dali spraviť ešte rôzne optimalizácie.

Pretože som sa nemal počas doterajšieho štúdia príležitosť stretnúť s touto problematikou, získal som pri vypracovávaní tejto práce úplne nové vedomosti o oblasti robotiky a fungovaní a programovaní rôznych senzorov na robotickom systéme ako aj nové vedomosti z oblasti umelej inteligencie.

Zdroje a použitá literatúra

[1] CORMEN, T. H., a kol.: *Introduction to Algorithms.* MIT Press, 2001.

[2] KENT, STEVEN L. *The First Quarter: A 25-year history of video games*. BWD Press, 2000.

ISBN 0-9704755-0-0.

[3] KENT, STEVEN L. *The Ultimate History of Video Games*. San Val Inc., 2001,

ISBN 0-613-91884-3.

[4] MAŘÍK, V., a kol. *Umělá inteligence (1)*. Praha : Academia, 1993,

ISBN 80-200-0496-3.

[5] NÁVRAT, P. a kol.: *Umelá inteligencia*. STU v Bratislave, 2002,
ISBN 80-227-1645-6.

[6] RUSSEL, S.; NORVIG, P.. *Artificial Intelligence: A Modern Approach*. 2. vyd. New Jersey, USA : Prentice Hall, 2003, ISBN 0-13-790395-2.

[7] Sokoban solver Rolling Stone
(online, pristupované dňa 6.1.2016)
https://webdocs.cs.ualberta.ca/~games/Sokoban/program.html

[8] VIRKKALA, T., *Solving Sokoban*, UNIVERSITY OF HELSINKI , 2011
(online, pristupované dňa 6.1.2016)
http://weetu.net/Timo-Virkkala-Solving-Sokoban-Masters-Thesis.pdf

[9] PETROVIČ, P., *Tatrabot - a mobile robotic platform for teaching programming*, Constructionism 2016