ROS - Move Base

Nicolas Limpert und Christian Schnieder

[1] Fachhochschule Aachen - University of Applied Sciences
[2] Robotik WS 2014 / 2015

Zusammenfassung. Dieser Bericht stellt eine Zusammenfassung der verschiedenen Komponenten des in ROS implementierten Move_Base-Pakets dar.
Das ROS-Package move_base bietet die Möglichkeit mit gegebener Karte und Anfangsposition ein örtliches Ziel mit einem Roboter zu erreichen, auf dem der Navigation Stack [1] in ROS vollständig implementiert ist, da die Move_Base lediglich eine Schnittstelle für den Benutzer darstellt. Diese Schnittstelle bietet für den Nutzer die Funktionalität einen Navigationsbefehl absetzen (beispielsweise über das Tool *RViz*) und diesen Befehl von der Base unter Berücksichtigung von Kollisionsvermeidung ausführen zu können.

1 Einleitung

Das Move_Base-Paket ist ein ROS-Paket, das dazu verwendet wird einem Roboter ein örtliches Ziel zu vermitteln und dieses Ziel in Kombination von mehreren ROS-Nodes (globalen- und lokalen Planer, globale und lokale costmap, etc.) zu versuchen, zu erreichen.
Das Ziel wird der Move Base in Form einer Action vermittelt, mit der Idee aus einer ROS-Message nach Möglichkeit eine Reihe von Fahrbefehlen unter Berücksichtigung von Kollisionsvermeidung, optimaler Pfadplanung (in Abhängigkeit von lokalem und globalem Planner) auszuführen.
Das Move_Base-Paket stellt dadurch einen essentiellen Teil des nav_cores in ROS dar, da es einem Benutzer bzw. einer Anwendung einfach ermöglicht ein gewünschtes örtliches Ziel zu erreichen, indem das Zusammenspiel zwischen den verschiedenen Komponenten des nav_cores koordiniert wird.

2 Motivation

Aufgrund der steigenden Komplexität in Robotersystemen möchte man die verschiedenen Teilaufgaben gemäß moderner Softwareentwicklungsverfahren gestalten bzw. wie in der objektorientierten Softwareentwicklung einer Instanz des Systems eine Aufgabe erteilen, die sich dann um die Erfüllung (oder Scheiterung) dieser Aufgabe kümmert.
Durch eine geschickte Gestaltung der verschiedenen Softwarekomponenten erhält man eine einfache Wart- und Erweiterbarkeit des Systems, die sich in einem

Robotersystem so äußern könnte dass man einem Teil der Software mitteilt dass man ein bestimmtes Ziel erreichen möchte und dieses Ziel dann von diesem Teil erfüllt wird.
Dies ermöglicht die Move_Base, in dem ihr mittels ROS-Message ein Ziel in Form einer geometry_msgs/PoseStamped Nachricht vermittelt wird, welches Ziel zu erreichen ist. Nachfolgend wird im Detail erklärt, welche Komponenten dabei wichtige Rollen spielen.

3 Komponenten

Move_base besteht aus folgenden Komponenten: [2]

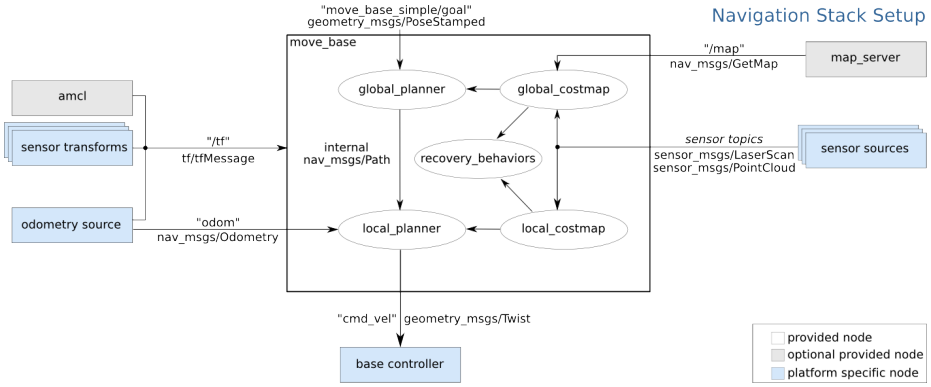

Abb. 1. Komponentenübersicht
[3]

Globaler Planer Dies kann ein beliebiger globaler Planer sein, solange er das Interface nav_core::BaseGlobalPlanner erfüllt. Er ist für die Pfadplanung innerhalb der Karte zuständig, führt also beispielsweise eine A* - Suche durch, um von einem gegebenen Anfangspunkt den gewünschten Endpunkt zu erreichen.

Lokaler Planer Dies kann ein beliebiger Planer sein, solange er das Interface nav_core::BaseLocalPlanner erfüllt. Aufgabe des lokalen Planers ist das Erreichen des nächsten Punkts den der globale Planer gegeben hat, bzw. das ausgeben von Fahrbefehlen zum erreichen dieses nächsten Punkts.

Globale Costmap Die globale Costmap, näher unter Punkt 4 beschrieben.

Lokale Costmap Die lokale Costmap, näher unter Punkt 4 beschrieben.

Recovery Behaviour Die Recovery Behaviours, die dazu dienen ein Verhalten für Probleme mit dem geplanten Pfad und der Dynamik der Umwelt zu definieren. Näher beschrieben unter Punkt 3.3.

Nachfolgend werden die einzelnen Komponenten beschrieben.

3.1 Globaler Planer

Der globale Planer erhält unter dem ROS-Topic *move_ base_ simple/goal* eine Message vom Typ *geometry_ msgs/PoseStamped*, die unter Berücksichtigung der globalen Costmap beispielsweise mithilfe von A* berechnet, welcher Pfad durch die einzelnen Knotenpunkte in der Karte zum Ziel führen.
Hierbei wird seitens der ROS-Wiki als grundliegende Implementationen *navfn* und der *carrot_ planner* aufgeführt:

navfn Über die jeweilige Karte wird ein Grid gelegt, durch das von der Anfangsposition aus bis zum Endpunkt eine Suche mittels Dijkstra durchgeführt wird, die den optimalen Pfad liefert, sofern er verfügbar ist.
Laut Diskussion wurde hier auf Dijkstra gesetzt, da er im Vergleich zu einem A* - Algorithmus eher optimale Pfade liefert. [6]

carrot_planner Wenn der Zielpunkt innerhalb eines Objekts liegt, wird entlang des Vektors vom Roboter zur Zielposition so lange zurück gegangen, bis ein Punkt ausserhalb des Objekts gefunden wurde, den der Roboter erreichen kann. Dadurch wird dem Roboter ermöglicht, das Ziel so nah wie möglich zu erreichen. Es wird hierbei also nicht viel geplant, sondern lediglich einem lokalen Planer ein einfacher Plan zur Verfügung gestellt. [7]

3.2 Lokaler Planer

Aufgabe des lokalen Planers ist das Erreichen des nächsten Punkts, der im vom globalen Planer übermittelten Plan steht. Grundliegend ist in ROS der *base_ local_ planner* implementiert:

base_local_planner Dieses Paket beinhaltet zwei verschiedene Implementierungen zur Ausführung eines vom globalen Planer gegebenen Pfads. Dazu zählt das *Trajectory Rollout* und der *Dynamic Window Approach (DWA)*, die folgendes Verfahren gemeinsam haben:

1. Ermittle verschiedene Bewegungsmuster die dem Roboter zur Verfügung stehen (dx, dy, dtheta)
2. Simuliere eine Bewegung für jedes dieser Bewegungsmuster um abzuschätzen, was beim Ausführen dieser Bewegung passieren würde wenn die Bewegung für einen bestimmten (kurzen) Zeitraum ausgeführt wird.
3. Bewerte jedes Bewegungsmuster, das in der Simulation ausgeführt wurde anhand von Werten wie Abstand zu anderen Objekten, Nähe zum Zielpunkt, Nähe an den globalen Pfad und Geschwindigkeit. Verwerfe ungültige Bewegungsmuster, also solche, die mit anderen Objekten kollidieren.
4. Nimm das Bewegungsmuster mit der höchsten Bewertung und gebe dieses als ausführbare Steuerbefehle (standardmäßig *geometry_ msgs::Twist* an Topic *cmd_ vel*) aus.

5. Wiederhole ab 1. Punkt.

Der konkrete Unterschied zwischen dem *Trajectory Rollout* und dem *Dynamic Window Approach (DWA)* besteht in der Erstellung der Bewegungsmuster. *Trajectory Rollout* probiert verschiedene mögliche Geschwindigkeiten über die gesamte gegebene Simulationszeit in Abhängigkeit von den für den Roboter möglichen Beschleunigungen.
DWA probiert die verschiedenen Geschwindigkeiten für nur einen Simulationsschritt (auch in Abhängigkeit von den Beschleunigungen).
DWA wird laut ROS-Wiki aufgrund seiner höheren Effizienz verwendet (es probiert innerhalb von kleineren möglichen Bewegungsmuster als *Trajectory Rollout*) [8] Zum *DWA* gibt es zusätzlich eine separate Implementierung in ROS: **http://wiki.ros.org/dwa_local_planner**.

3.3 Recovery Behaviours

Die Recovery Behaviours sind ein optionales Feature der Move_Base und eignen sich für einen Umstand, in dem der Roboter zwar durch den globalen Planer einen gültigen Plan erhalten hat, aber diesen, beispielsweise aufgrund der Dynamik in der Umwelt, nicht länger verfolgen kann und deswegen einen neuen Pfad berechnen muss.

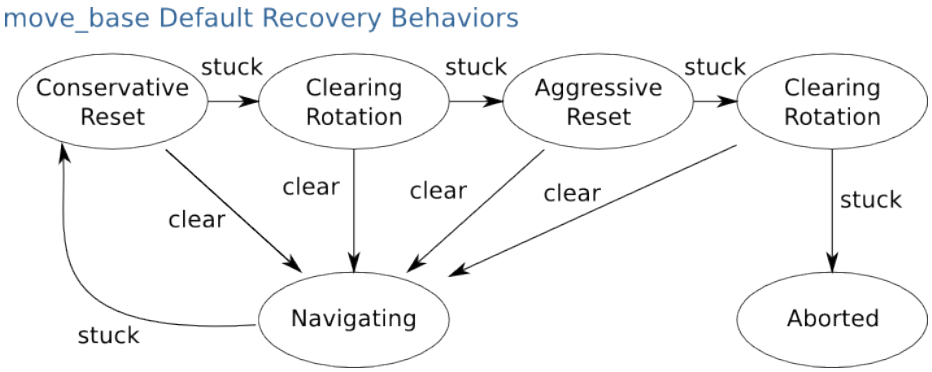

Abb. 2. Recovery Behaviours
[4]

Dabei wird folgendes Schema verwendet: [5]

Navigating Solange die Navigation ohne Probleme weiter läuft (globaler Pfad ist verfolgbar), fahre mit der Navigation fort.

Conservative Reset Wenn der Roboter feststeckt werden zu Beginn alle Objekte ausserhalb eines vom Benutzer festgelegten Bereichs entfernt. Danach wird nach Möglichkeit eine Rotation auf der Stelle durchgeführt, um den Raum innerhalb der costmaps zu klären, damit ein neuer Pfad gefunden werden kann.

Aggresive Reset Wenn das Conservative Reset nicht geholfen hat, wird beim Aggressive Reset die costmap innerhalb eines festgelegten Bereichs vollständig zurückgesetzt und anschließend eine erneute Clearing Rotation durchgeführt.

Aborted Wenn auch der vorige Schritt fehlschlägt wird davon ausgegangen dass aus der aktuellen Situation kein Ausweg besteht und die Navigationsaufgabe wird abgebrochen.

4 Costmap

Das costmap_2d Paket [9] bietet eine konfigurierbare Struktur in Form eines *occupancy grids*. Costmap benutzt Sensorinformationen und Informationen von der statische Karte um neue Objekte in Form eines costmap_2d::Costmap2DROS Objektes zur Verfügung zu stellen. Dieses Objekt bietet eine rein zweidimensionale Schnittstelle. Das bedeutet, dass Objekte nur zweidimensional dargestellt werden können. Dies dient der vereinfachten Planung in der Ebene.
Seit der ROS-Version Hydro sind die Methoden zur costmap-Konfiguration frei konfigurierbar. Jede Funktionalität ist in einer Ebene abgelegt. So ist beispielweise die statische Karte eine Ebene und die Hindernisse sind eine andere. Standardmäßig sind die Hindernisse in der entsprechenden Ebene in 3D abgelegt. Dieses ermöglicht einen effizienteren Umgang mit ihnen beim markieren und freigeben des Raumes.
Die Hauptschnittstelle ist costmap_2d::Costmap2DROS, die viele ROS Funktionalitäten enthält. Sie enthält costmap_2d::LayeredCostmap welches es ermöglicht die einzelnen Ebenen im Auge zu behalten. Jede Ebene ist in der LayeredCostmap enthalten, jedoch kann jede Ebene einzeln implemeniert werden. Die Funktionalität wird über die costmap_2d::Costmap2D Klasse zur Verfügung gestellt.

4.1 Markieren und Löschen

Die Costmap aktualisiert sich ständig mit Hilfe von Sensortopics über ROS. Jeder Sensor benutzt entweder *mark* (fügt Hindernis-Informationen zur costmap hinzu), *clear* (entfernt Hindernis-Informationen aus der costmap) oder beides. Zum Markieren wird in das Array in die entsprechende Zelle die Kosten eingetragen. Ein Löschvorgang besteht jedoch in jeder einzelnen Beobachtung aus Raytracing durch ein Gitter von dem Ursprung des Sensors nach außen. Wenn eine dreidimensionale Struktur verwendet wird, werden Hindernisinformationen aus jeder Ebene in zwei Dimensionen umgerechnet, wenn diese in die costmap eingetragen werden.

4.2 Occupied, Free, und Unknown Space

Während jede Zelle in der costmap Werte von 0 bis 255 haben kann, hat die zugrunde liegende Struktur nur drei unterschiedliche Werte. Jede Zelle kann entweder als frei, benutzt oder unbekannt markiert werden. Jeder Status hat einen bestimmten Wert. Die Werte für occupied Zellen sind hinterlegt in costmap_2d::LETHAL_OBSTACLE, die Kosten für unknown Zellen sind hinterlegt in costmap_2d::NO_INFORMATION und die Zellen für free Space sind hinterlegt in costmap_2d::FREE_SPACE.

4.3 Map Updates

Die costmap wird in Zyklen mit einer Frequenz, definiert in *update_frequency*, aktualisiert. In jedem Zyklus gibt es neue Sensordaten, die ein erneutes Markieren und Löschen in der darunterliegenden Struktur der Belegungs-costmap erfordern. Diese Struktur wird in die costmap, wo die entsprechenden Kostenwerte wie oben beschrieben zugeordnet sind, projiziert. Danach wird auf jedes Hindernis mit seinen Zellen eine Operation durchgeführt, die sich Inflation nennt und mit Kosten belegt die in costmap_2d::LETHAL_OBSTACLE wiederzufinden sind. Diese besteht aus Vermehrungskostenwerte nach außen von jeder besetzten Zelle aus einem Benutzer - spezifizierten Reifenradius. Die Details dieser Inflation Verfahrens werden nachstehend beschrieben.

4.4 tf

Um Daten aus Sensorquellen in das costmap einzufügen, benutzt das costmap_2d::Costmap2DROS Objekt tf. Genauer gesagt, wird davon ausgegangen, dass alle Transformationen zwischen den vom global_frame Parameter, dem robot_base_frame Parameter und Sensorsignale auf dem neuesten Stand sind. Der Parameter transform_tolerance legt die maximale Latenz zwischen diesen Transformationen fest. Wenn der tf Baum nicht mit der erwarteten Rate aktualisiert wird, dann wird der Navigation Stack des Roboters gestoppt.

4.5 Inflation

Die Inflation ist der Prozess der Verbreitung Kostenwerte aus besetzten Zellen verringert um den Abstand. Zu diesem Zweck definieren wir 5 spezifische Symbole für costmap Werte.

Lethal Kosten bedeutet, dass es ein tatsächliche Hindernis in einer Zelle gibt. Würde sich das Zemtrum des Roborts dort befinden, gäbe es eine Kollision.

Inscribed Kosten bedeutet, dass eine Zelle weniger als der Roboterradius entfernt ist.

Possibly circumscribed Kosten sind ähnlich den **inscribed** Kosten, nur das hier die Roboterkonturen zu Grunde gelegt werden.

Freespace Kosten werden als Null angenommen. Eine freie Roboterbewegung ist möglich.

Unknown Kosten bedeutet, dass keine Informationen zu einer bestimmten Zelle gibt. Der Benutzer des costmap kann dies interpretieren, wie er will.

Alle anderen Kosten liegen zwischen **Freespace** und **Possibly circumscribed** abhängig von der Entfernung zur **lethal** Zelle und der zugrundeliegenden benutzerdefinierten Funktion.

4.6 Map Types

Es gibt zwei Möglichkeiten, um ein costmap_2d::Costmap2DROS Objekt zu initialisieren:

Die erste ist die Nutzung einer benutzergenerierten statischen Karte. In diesem Fall wird die costmap so initialisiert, dass die Breite, Höhe und Hindernisinformationen mit der statischen Karte übereinstimmen. Diese Konfiguration wird normalerweise in Verbindung mit einem Lokalisierungssystem verwendet.

Die zweite Möglichkeit eine costmap_2d::Costmap2DROS Objekt zu initialisieren ist die Breite und Höhe zu definieren und den rolling_window Parameter auf *true* zu setzten. Der rolling_window Parameter halten den Roboter in der Mitte der costmap wenn diese sich durch die Welt bewegt.

5 Action API

Der move_base Knoten stellt eine Implementierung des SimpleActionServer (siehe actionlib Dokumentation) zur Verfügung der die Ziele der *geometry_msgs/PoseStamped* Nachrichten enthält. Die Kommunikation mit den move_base Knoten ist direkt über ROS möglich. Es wird aber empfohlen die Ziele über den SimpleActionClient zu schicken, sofern eine Nachverfolgung des Status gewünscht wird.

move_base/goal (move_base_msgs/MoveBaseActionGoal)
Ein Ziel das move_base in der Welt verfolgen soll.

move_base/cancel (actionlib_msgs/GoalID)
Anfrage um die verfolgung eines Ziels abzubrechen.

5.1 Action Published Topics

move_base/feedback (move_base_msgs/MoveBaseActionFeedback)
Enthält die aktuelle Position der Basis in der Welt

move_base/status (actionlib_msgs/GoalStatusArray)
Enthält Statusinformationen der Ziele die an move_base geschickt werden.

move_base/result (move_base_msgs/MoveBaseActionResult)
Rückgabewert ist leer für move_base action.

6 Topics

6.1 Subscribed Topics

move_base_simple/goal (geometry_msgs/PoseStamped)
Bietet ein non-action interface für move_base für Benutrzer bei denen die Nachverfolgung der Aktionen nicht relevant ist.

6.2 Published Topics

cmd_vel (geometry_msgs/Twist)
Fortlaufende Bewegungsbefele die durch eine mobile Base ausgeführt werden sollen.

7 Services

make_plan (nav_msgs/GetPlan)
Ermöglicht einem externen Benutzer einen Plan zu einem Ziel zu bekommen ohne das move_base diesen verfolgt.

clear_unknown_space (std_srvs/Empty)
Ermöglicht einen externen Benutzer den Bereich um den Roboter auf leer zu setzen. Dieses kann verwendet werden wenn der Roboter in einen neue Umgebung gesetzt wird.

clear_costmaps (std_srvs/Empty)
Erlaubt einen externen Benutzer Objecte aus der costmap zu entfernen. Da dieses zu Kollisionen führen kann, sollte diese Funktion mit Vorsicht benutzt werden. Sie kann dennoch hilfreich sein, wenn es Fehlern in der costmap gibt.

8 Parameter

Nachfolgend werden Parameter an die Move_Base aufgelistet, die entweder beim Starten der Move_Base beispeilsweise in einem Launch-File oder über den ROS-Parameterserver gesetzt werden.

base_global_planner (string, default: navfn/NavfnROS")
Name des Plugins das als Globaler Planer mit move_base benutzt wird. VGL. Abschnitt "Komponenten / Globaler Planer": 3.1

base_local_planner (string, default: "base_local_planner/TrajectoryPlannerROSFF-or 1.1+ series)
VGL. Abschnitt "Komponenten / Lokaler Planer": 3.2

recovery_behaviors (list, default: [name: conservative_reset, type: clear_costmap_recovery/ClearCostmap
name: rotate_recovery, type: rotate_recovery/RotateRecovery, name: aggressive_reset, type: clear_costmap_recovery/ClearCostmapRecovery])
VGL. Abschnitt "Komponenten / Recovery Behaviours": 3.3

controller_frequency (double, default: 20.0)
Die Rate in Hz in der die Fahrbefehle an den Basiskonten gesendet werden.

planner_patience (double, default: 5.0)
Wert wie lange in Sekunden der Planer versucht einen Plan zu finden bis space-clearing Operationen eingeleitet werden (siehe **Recovery Behaviours** 3.3).

controller_patience (double, default: 15.0)
Wert wie lange in Sekunden der Kontroller auf richtige Komandos wartet bis spaceclearing Operationen eingeleitet werden.

conservative_reset_dist (double, default: 3.0)
Abstand von Hindernissen in Metern zum Roboter die aus der costmap entfernt werden wenn versucht wird Platz in der costmap zu schaffen.

recovery_behavior_enabled (bool, default: true)
Ein und ausschalten der move_base recovery behaviors.

clearing_rotation_allowed (bool, default: true)
Erlaubt oder verbietet Rotationen um die eigene Achse beim Versuch von clear out space Hinweis: Diese Parameter wird nur beim Standard recovery behaviors benutzt.

shutdown_costmaps (bool, default: false)
Gibt an ob die costmap ausgeschaltet wird wenn die move_base auf inaktiv gesetzt ist.

oscillation_timeout (double, default: 0.0)
Wert in Sekunden bis das recovery behaviour ausgelöst wird.

oscillation_distance (double, default: 0.5)
Strecke in Metern die der Roboter zurücklegen muss um sicher zu sein nicht zu schwingen. Setzt den Timer oscillation_timeout zurück.

planner_frequency (double, default: 0.0)
Frequenz in Hertz in der die Globale Planer Schleife ausgeführt wird. 0.0 führt den Globalen Planer nur bei einem neuen Ziel aus oder wenn der lokale Planer eine unmögliche Route meldet.

9 Beispielhafte Implementierung an einem Rover

Nachfolgend wird ein Beispiel mithilfe der vom Maskor-Institut der Fachhochschule Aachen gestellten Rover aufgeführt, mit dem die Verwendung der *move_base* möglich ist. Dazu nimmt man für den globalen und lokalen Planer andere Pakete, die sich eher für Roboter mit einer Lenkung wie der eines Kraftfahrzeugs (Ackermann) eignen.

9.1 Problem: Ackermann im Vergleich zu (nicht-) holonomen Robotern

Ein Roboter, der ein Ackermann-Fahrgestell besitzt unterscheidet sich von anderen mobilen Robotern grundliegend darin dass er sich einerseits nicht auf der Stelle drehen kann und sich auch nicht seitlich bewegen kann.
Die oben genannten Implementationen der globalen und lokalen Planer gehen aber davon aus dass der Roboter sich zumindest auf der Stelle drehen kann.
Dies erschwert die Möglichkeit, den *ROS Navigation Stack* mit Robotern zu verwenden, die Ackermann-getrieben sind. [10]

Lösung Für die Pfadplanung (bzw. Aktionsplanung) existiert eine von der Carnegie Mellon University (Dr. Maxim Likhachev [14]) entwickelte Bibliothek, die sich (*SBPL - Search Based Planning Library* [11]) nennt und es, neben anderen Features, erlaubt unter Verwendung des *SBPL Lattice Planners* eine Folge von Aktionen durchzuführen, die einem Ackermann-getriebenen Roboter ermöglichen ein Ziel zu erreichen.
In ROS bedeutet das dass man der Move_Base als globalen Planer den *SBPL Lattice Planner* gibt und als lokalen Planer einen angepassten *base_local_planner*.
Der *SBPL Lattice Planner* kommt hier als globaler Planer zur Verwendung, da er die Möglichkeit darstellt, einen Pfad zu berechnen der den Aktionenraum des gegebenen Roboters berücksichtigt und nicht darüber hinaus geht. Der berechnete Pfad wird also so erstellt dass er von einem Roboter mit eingeschränkter Bewegungsfähigkeit (im Vergleich zu einem Roboter, der sich auf der Stelle drehen kann) ausgeführt werden kann.

9.2 Globaler Planer

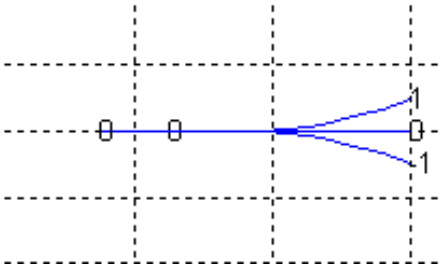

Abb. 3. Motion Primitive (Matlab) [13]

Wie bereits beschrieben kommt hier der SBPL Lattice Planner zum Einsatz.
Damit der Lattice Planner weiß, welche Möglichkeiten er für die Umsetzung seines Plans hat, benötigt er neben der Karte (die nach wie vor eine costmap ist) eine Reihe von Motion Primitives. Diese werden nachfolgend beschrieben:

Motion Primitives sind im SBPL-Kontext Beschreibungen für Aktionen, die ein Roboter in einem Zustand durchführen kann.
Am Beispiel des Rovers wäre das also entweder eine Geradeausfahrt oder eine Fahrt stark oder leicht zur einen oder anderen Seite eingeschlagen.
Jedes Motion Primitive besitzt einen Anfangszustand und kann mehrere Endzustände haben. Dies wird benötigt, damit eine Verknüpfung der Motion Primitives aneinander erfolgen kann bzw. diese als Graphen betrachtet werden können. SBPL bietet auch die Möglichkeit die Bewegung eines Roboterarmes zu planen bzw. auch kollisionsfrei durchzuführen, dies soll hier allerdings nicht weiter erläutert werden.

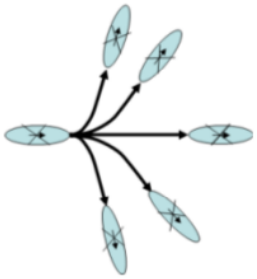

Abb. 4. Motion Primitives im Graphen [13]

Pfaderstellung SBPL verknüpft Motion Primitives, legt mit den verknüpften Motion Primitives einen Graph über die Umwelt und streicht, wie im Bild gezeigt, Knoten, die in einem Hindernis liegen oder eine Kollision auslösen würden bzw. (laut costmap, Abschnitt 4) für den Roboter unerreichbar sind.
Im Anschluss daran kann innerhalb von SBPL eine Graphensuche über die verschiedenen aneinander geknüpften Motion Primitives geführt werden, die einen gültigen Pfad liefert (solange dieser existiert) und der die Bewegungsmöglichkeiten des Roboters berücksichtigt.
Dabei können theoretisch Schleifen entstehen. Es wird aber später im Bereich der Algorithmen zur Graphensuche gezeigt, warum diese keine Schwierigkeit darstellen. Die in SBPL implementierten Algorithmen für diese Graphensuche sind wie folgt:

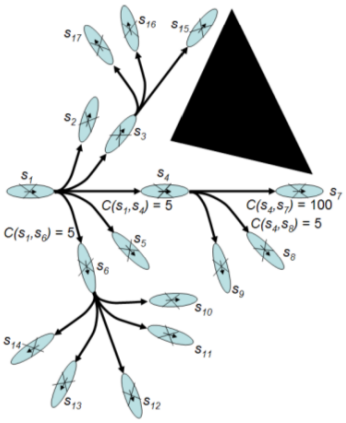

Abb. 5. Motion Primitive
[13]

ARA* - Anytime Repairing A* funktioniert wie der A* - Algorithmus, nur dass die Heuristik-Funktion zusätzlich eine Gewichtung ϵ erhält, die in mehreren Suchschritten verkleinert wird und die Daten der Open-List nicht jedes mal neu erstellt werden müssen sondern wieder verwendet werden können.

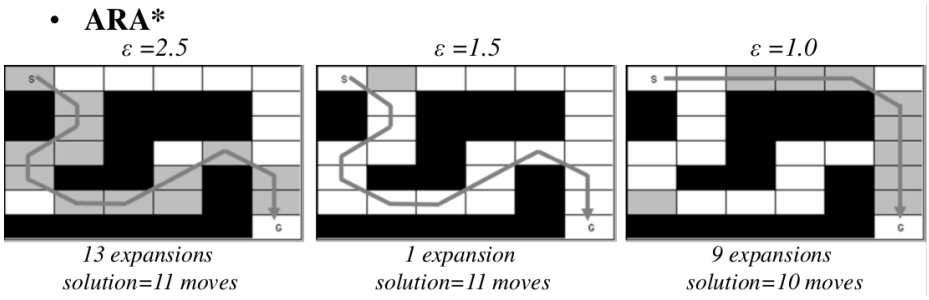

Abb. 6. ARA*
[13]

Dadurch kann ARA* anfangs eine Lösung liefern die vielleicht nicht der optimalen Lösung entspricht aber in der Regel geringeren Speicherbedarf hat und in kürzerer Zeit erreichbar ist.
Dies bietet sich an, da in der Praxis oft Resultate zufriedenstellend sind, die

nicht der besten Lösung entsprechen aber für das Problem ausreichen.
Bei weiteren Iterationen wird ϵ verkleinert, sodass die Chance auf eine bessere Lösung steigt. TODO In ARA* wird zur Beeinflussung der Gewichtung ein ϵ

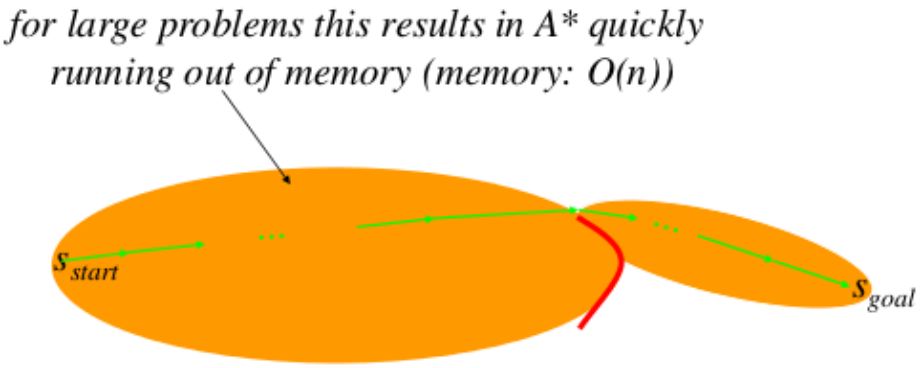

Abb. 7. A* im Vergleich zu weighted A*
[13]

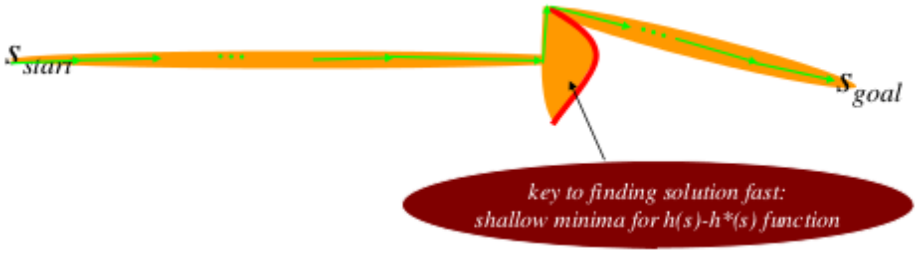

Abb. 8. Weighted A* im Vergleich zu A*
[13]

genutzt (weiter unten beschrieben). Die Abbildungen 7 und 8 zeigen A* und Weighted A*, das ein essentieller Bestandteil von ARA* ist, im Vergleich.
Dabei wird deutlich dass A*, wie vorher beschrieben, zwar einen optimalen Pfad liefert, dafür aber unter Umständen viel Speicher benötigt, während ARA* eine Gewichtung einfügt (weighted A*) und neben der Gewichtung ϵ bereits geöffnete Knoten erneut verwenden kann.

Die Formel zur Bestimmung von f bei weighted A* sieht wie folgt aus (auch als Wiederholung der Eigenschaften von A*):
$f(s) = g(s) + \epsilon h(s), \epsilon \geq 1.$

- s ist die Zelle bzw. der Zustand.
- f ist der Funktionswert der Zelle s, der dessen Reihenfolge in der von A* genutzten Priority Queue bestimmmen soll.
- g ist die bisherige Distanz vom Anfangsknoten zum aktuellen Knoten
- ϵ ist die Gewichtung für die Heuristik $h(s)$. Ein $\epsilon = 1$ entspricht dem klassischen A*.
- h ist die Heuristik. Diese darf die echte Distanz nicht überschätzen, sodass gilt: $\forall s : h(s) \leq c * (s, s_{goal})$ wobei c die minimalen Kosten von s zu s_{goal} sind.

Anytime D* Neben der Implementierung des Anytime Repairing A* bietet SBPL zur Graphensuche alternativ eine inkrementelle Variante des A* Algorithmus, dem D*, die hier allerdings nicht weiter erläutert wird, da sie nicht in dem Beispiel verwendet wird.

9.3 Lokaler Planer

Der lokale Planer ergibt sich im wesentlichen aus dem bereits in Punkt 3.2 beschriebenen.
Allerdings werden im **TrajectoryPlanner** einige Anpassungen gemacht:

- Der Roboter darf sich nicht mehr auf der Stelle drehen. Trajektorien, die das voraussetzen, sind also in *trajectory_planner.cpp* entsprechend anzupassen.
- Zusätzlich müssen dem lokalen Planer die Rückwärtsbewegungen hinzugefügt werden, sodass Trajektorien gemäß dem globalen Planer (SBPL) funktionieren. TODO

9.4 Globale und lokale Costmap

Die lokale und globale Costmap werden mithilfe eines Mapservers bereitgestellt. Verfahren wie SLAM sind in der frühen Entwicklungsphase noch nicht implementiert, da es vorrangig um die Fähigkeit geht, die Move_Base verwenden zu können.

9.5 Recovery Behaviours

Recovery Behaviours werden wegen der frühen Entwicklungsphase nicht verwendet, sind aber ggf. eine Option um das Verhalten des Roboters weiter zu verbessern.

Literatur

[1] Navigation, “http://wiki.ros.org/navigation”
[2] Move_Base, “http://wiki.ros.org/move_base”
[3] Move_Base-Components Image
“http://wiki.ros.org/move_base?action=AttachFile&do=view&target=overview_tf.png”
[4] Recovery Behaviours Image
“http://wiki.ros.org/move_base?action=AttachFile&do=get&target=recovery_behaviors.png”
[5] Rotate Recovery “http://wiki.ros.org/rotate_recovery”
[6] navfn, “http://wiki.ros.org/navfn”
[7] CarrotPlanner, “http://wiki.ros.org/carrot_planner”
[8] base_local_planner, “http://wiki.ros.org/base_local_planner”
[9] costmap_2d, “http://wiki.ros.org/costmap_2d”
[10] Ackermann Group, “http://wiki.ros.org/Ackermann
[11] SBPL, “http://sbpl.net”
[12] SBPL Lattice Planner, “http://wiki.ros.org/sbpl_lattice_planner”
[13] SBPL ROS School,
“http://www.ros.org/wiki/Events/CoTeSys-ROS-School?
action=AttachFile&do=get&target=robschooltutorial_oct10.pdf”
[14] Dr. Maxim Likhachev, “http://www.cs.cmu.edu/~maxim/”